Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

Dissertação de Mestrado

Mestrado em Engenharia Informática

Byzantium: Contribuições para o desenvolvimento de um sistema de replicação de bases de dados tolerante a falhas bizantinas

Rui Alexandre Bom Garcia (aluno n.º 26173)

2º Semestre de 2008/09

30 de Novembro de 2009

Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

Dissertação de Mestrado

Byzantium: Contribuições para o desenvolvimento de um sistema de replicação de bases de dados tolerante a falhas bizantinas

Rui Alexandre Bom Garcia (aluno n.º 26173)

Orientador: Prof. Doutor Nuno Preguiça

Trabalho apresentado no âmbito do Mestrado em Engenharia Informática, como requisito parcial para obtenção do grau de Mestre em Engenharia Informática.

2º Semestre de 2008/09

30 de Novembro de 2009

Agradecimentos

Este trabalho foi suportado por uma bolsa de investigação da FCT/MCTES, projecto PTDC/ EIA/74325/2006.

Resumo

As bases de dados são aplicações extremamente utilizadas hoje em dia, e por isso, é importante que cumpram determinados requisitos, nomeadamente, correcção, disponibilidade e desempenho. Uma aproximação para atingir estes objectivos passa por replicar o estado da base de dados em diferentes localizações, sendo necessário manter a coerência entre as diferentes réplicas. Esta aproximação tem a possibilidade de continuar a fornecer o serviço mesmo em caso de falha de uma parte dos componentes do sistema.

A maioria das soluções existentes procura resolver o problema assumindo apenas o modelo de falhas *fail-stop*, sem considerar falhas bizantinas. As falhas bizantinas provocam o comportamento arbitrário dos componentes, normalmente associadas a erros de implementação ou situações imprevistas como falhas de *hardware* ou máquinas controladas por atacantes.

Neste trabalho desenvolveu-se a segunda versão do Byzantium, um sistema *middleware* de replicação de bases de dados tolerante a falhas bizantinas com o modelo de isolamento *Snapshot Isolation*. Além de uma nova implementação do protótipo, esta nova versão propõe novos algoritmos que introduzem as seguintes novas funcionalidades. Primeiro, um novo mecanismo eficiente de propagação das operações de uma transacção para as várias réplicas no sistema, durante a sua execução. Este mecanismo pretende diminuir a complexidade da operação *commit*. Segundo, a utilização de um menor número de réplicas para executar as transacções apenas com leituras. Esta decisão permite diminuir a carga das réplicas. Terceiro, a execução imediata das operações de leitura. Esta aproximação faz com que as transacções de leitura obtenham o resultado da operação de *commit* localmente no cliente, sem necessidade de contactar o servidor.

Os resultados obtidos na avaliação do protótipo mostram que estes algoritmos têm alguma penalização face a uma solução não tolerante a falhas com apenas um servidor para transacções de leitura e escrita. Adicionalmente, revela um desempenho superior para transacções de leitura quando comparado com uma solução simples não tolerante a falhas. Quando as mesmas optimizações introduzidas no Byzantium são usadas na execução das operações na solução não tolerante a falhas, o desempenho do protótipo implementado é ligeiramente inferior.

Palavras-chave: Replicação bizantina, bases de dados, *Snapshot Isolation*, *middleware*.

Abstract

Database systems are widely used for supporting a large number of different applications. These systems must meet some requirements, such as correction, availability and performance. Replication can be used to achieve these goals by keeping the database state at several different locations. This approach makes it possible to continue to provide the service even in the presence of failed replicas, but requires a solution to keep the replicas coherent.

Most existing solutions address the problem assuming the fail-stop failure model, without considering byzantine faults. Byzantine faults are those that lead to component arbitrary behavior, usually associated with implementation errors or unforeseen situations like hardware faults or computers controlled by attackers.

This work presents the second version of Byzantium system that intends to create a byzantine fault-tolerant database replication middleware that provides Snapshot Isolation. This new version improves in several directions. More concretely, besides some minor improvements in the system, the objective is to develop, test and explore the following three solutions. First, improving read-only transactions by propagating the queries immediately in order to reduce que cost of the commit operation. Second, allowing concurrent execution of multiple read-only transactions using different sets of replicas by reduction the number execution replicas for read operations. Third, executing read operations during the execution of transactions. This solution allows the immediate return of commit operations in read-only transactions.

After the evaluation of the implemented prototype, we can conclude that Byzantium has some overhead when compared with a simple non fault tolerant solution, for a read-write workload. Also, results show that Byzantium has a superior performance when using a read-only workload. On the other hand, when using an optimized proxy, Byzantium prototype shows a slightly inferior performance.

Keywords: Bizantine replication, databases, *Snapshot Isolation*, *middleware*

Conteúdo

1. Introdução

Neste capítulo faz-se uma introdução ao trabalho realizado nesta dissertação. Após uma introdução geral, descreve-se o contexto do trabalho e a aproximação tomada, assim como se descrevem as contribuições efectuadas.

1.1 Introdução Geral

As bases de dados são aplicações informáticas bastante utilizadas hoje em dia, e das quais dependem um elevado número de serviços. Seja em instituições bancárias ou apenas para uso pessoal, as bases de dados abrangem praticamente todas as áreas de *software*. As bases de dados são o método preferencial para arquivar informação nas aplicações de grande escala, onde é necessária coordenação nos acessos de vários utilizadores. Assim, são várias as situações em que as bases de dados são fundamentais, sendo importante que funcionem de acordo com o esperado. Dada a escala das aplicações em que são usadas, é importante que disponibilizem um desempenho aceitável para a aplicação. Por exemplo, uma operação bancária deve demorar apenas alguns segundos a processar. Além do desempenho é fundamental que seja garantida a correcção das operações. A juntar às anteriores propriedades, é importante que os dados se encontrem disponíveis sempre que necessário. Um sistema que garanta um bom desempenho, correcção e disponibilidade dos dados mesmo na presença de falhas aproxima-se do sistema ideal.

A correcção dos sistemas de bases de dados é normalmente definida pelas propriedades ACID [22]. O modelo ACID é um dos conceitos mais importantes e mais antigos na teoria de bases de dados definindo quatro propriedades que os sistemas de gestão de bases de dados devem procurar garantir: atomicidade, consistência, isolamento e durabilidade. A atomicidade refere-se à semântica de modificações, que consiste numa política de "tudo ou nada". Uma transacção diz-se atómica se todas as modificações são executadas ou nenhuma. A consistência é garantida se apenas forem efectuadas escritas válidas. Uma transacção garante a consistência se fizer evoluir a base de dados de um estado coerente para outro. O isolamento é a propriedade que garante que os efeitos de transacções simultâneas não causam impacto entre elas, isto é, uma transacção não deve observar o efeito de uma transacção concorrente. Por último, entende-se por durabilidade a capacidade que o sistema de gestão de bases de dados possui em assegurar

que os efeitos de uma transacção bem sucedida são salvaguardados em caso de falhas.

Com vista a aumentar o desempenho de um sistema, é frequente existir um relaxamento nestas propriedades. Em particular, é nesse sentido que foram definidos vários níveis de isolamento. O nível de isolamento *Serializable* (definido no ANSI SQL) encontra-se num dos extremos desse espectro, no qual se dá prioridade máxima à correcção, uma vez que não é permitida qualquer interacção entre transacções concorrentes sendo o resultado final equivalente a uma execução sequencial. Este é o nível de isolamento mais restrito e o único que assegura correcção total. Um pouco menos forte é o nível de isolamento implementado pelo modelo *Snapshot Isolation*[1] [4] (SI), que permite a interferência entre transacções de leitura e de escrita. Uma situação em que o nível de isolamento SI permite a interacção entre transacções é a seguinte: uma transacção T1 lê x e y que se encontram de acordo com uma condição C, e de seguida uma transacção T2 lê x e y, escreve x, respeitando C, e termina com sucesso. Por fim, T1 escreve y, respeitando C (com base no valor observado de x). Esta ordenação de operações possibilita a violação de uma restrição entre x e y, apesar de se encontrar de acordo com o modelo de isolamento SI.

O SI tem características de interesse que fazem com que seja um dos níveis de isolamento mais utilizados, nomeadamente o facto de as transacções de leitura nunca provocarem o bloqueio de qualquer outra transacção e de as transacções de leitura nunca bloquearem.

Em geral, o nível de isolamento a utilizar deve depender do programa criado e das garantias que se pretendem assegurar. É possível desenhar programas correctos a funcionar em níveis de isolamento fracos desde que se efectue algum controlo sobre as sequências de transacções que são produzidas. Por outro lado, pode até nem ser necessário utilizar um nível de isolamento forte, uma vez que existem aplicações que funcionam correctamente mesmo que ocorra interferência entre as transacções (por exemplo, aplicações que realizem operações estatísticas ou aproximações).

Devido à sua dimensão e complexidade, os sistemas de bases de dados estão sujeitos com elevada probabilidade a erros de implementação o que pode levar a comportamentos arbitrários [13, 23]. Além desse problema, estes sistemas estão sujeitos à acção por parte de atacantes

[1] Numa base de dados que utiliza esta técnica, do ponto de vista lógico, uma transacção T obtém no início da sua execução o último estado da base de dados, contendo os resultados de todas as transacções que fizeram *commit* antes da transacção T iniciar. Aquando do *commit* de T, o sistema verifica se o conjunto de escritas da transacção T não entra em conflito com os conjuntos de escritas das transacções que efectuaram *commit* após o início de T. Se a intersecção dos conjuntos de escrita for não vazia, a transacção T é abortada. Caso contrário, a transacção termina com sucesso.

que procuram comprometer a segurança e a correcção dos dados. Finalmente, estes sistemas estão sujeitos a erros de configuração que podem levar a um comportamento incorrecto [18]. Um sistema que apresente este tipo de erros ou comportamento continuando a funcionar mas devolvendo resultados incorrectos, diz-se que apresenta uma falha bizantina [16]. Dada a relevância das bases de dados, é importante desenvolver sistemas com a capacidade de tolerar este tipo de falhas, garantindo a correcção do sistema na presença de alguns nós falhados.

1.2 Contexto

Este trabalho enquadra-se no contexto do projecto Byzantium [20] cujo objectivo é desenvolver um sistema *middleware* eficiente de replicação de bases de dados que ofereça tolerância a falhas bizantinas e disponibilize o modelo de isolamento *Snapshot Isolation* (SI). Anteriormente foi desenvolvida uma primeira versão que pretendia avaliar a viabilidade da criação de um sistema com estas características. Este protótipo foi implementado com recurso ao sistema de replicação bizantina PBFT [5] sem modificações e utiliza sistemas de bases de dados *off-the-self*. Estes sistemas devem implementar nativamente o nível de isolamento *Snapshot Isolation* e suportar *savepoints*[2].

Apesar de não se encontrar implementado na versão inicial, a arquitectura do sistema Byzantium permite alterações no sentido de abstrair a implementação dos sistemas de gestão de bases de dados utilizados tornando possível a utilização de diferentes sistemas. A utilização de diferentes sistemas faz aumentar a independência (ou diminuir a correlação) entre as falhas bizantinas causadas por erros no *software*. Tanto a primeira versão do sistema, assim como a versão implementada nesta dissertação, encontram-se a funcionar apenas com um sistema, o PostgreSQL[3].

Como é ilustrado pela Figura 1.1, o sistema Byzantium é composto por um conjunto de réplicas de dimensão n=3f+1 onde f é o número máximo de réplicas que podem falhar simultaneamente e por um número finito de clientes. Nos clientes existe um componente responsável pelo tratamento dos pedidos e das respostas que abstrai a natureza replicada da solução e oferece à aplicação do utilizador uma interface JDBC. Do ponto de vista do utilizador, o sistema aparenta ser um sistema de gestão de bases de dados convencional.

[2] Um *savepoint* permite ao programador declarar um ponto na transacção para o qual é possível fazer *rollback* mais tarde.

[3] `http://www.postgresql.org/`

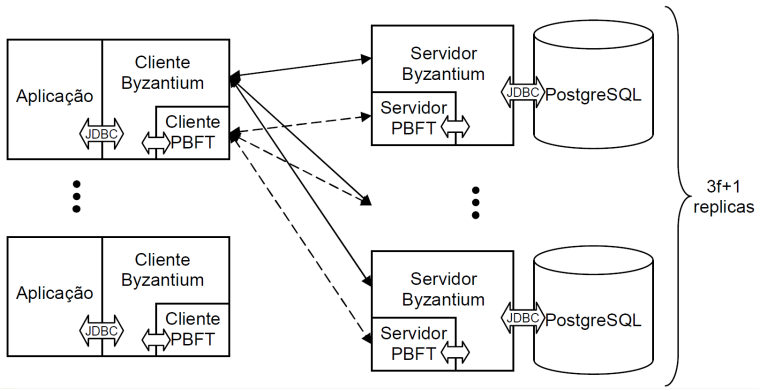

Figura 1.1 Arquitectura da primeira versão do sistema Byzantium (de [20]).

No lado das réplicas existe um componente que gere os pedidos recebidos e executa as operações no servidor de base de dados. A execução das transacções é efectuada com recurso a uma biblioteca que implementa o algoritmo de replicação bizantina PBFT. Esta biblioteca é tratada como uma caixa negra. Do ponto de vista do PBFT, o *middleware* actua como cliente (no lado do cliente) e como serviço replicado (no lado da réplica).

Neste sistema, as transacções são processadas da seguinte maneira: no início e no fim de cada transacção é utilizada uma operação tolerante a falhas bizantinas (*begin_trx*, *commit* e *rollback*) com o objectivo de ordenar totalmente as alterações ao estado das várias réplicas. A execução da operação *begin_trx* através do PBFT garante que a transacção executa no mesmo estado da base de dados em todas as réplicas. Submeter a operação *commit* (e *rollback*) pela biblioteca de replicação garante que as réplicas concluem as transacções pela mesma ordem nas diferentes réplicas. No início de cada transacção é seleccionada aleatoriamente uma réplica responsável pela transacção que trata todas as operações submetidas. Entre o início e o fim de uma transacção, o cliente pode submeter uma sequência de operações de leitura e escrita. Estas operações são executadas directamente na réplica primária e o seu resultado é devolvido imediatamente. No momento do *commit*, o cliente envia as operações e os seus resultados para as diferentes réplicas através da biblioteca de replicação bizantina. As réplicas executam as operações e verificam a correcção da execução inicial.

1.3 Solução Apresentada

Neste trabalho desenvolveu-se a segunda versão do sistema Byzantium, com o objectivo de melhorar o desempenho geral da solução inicial. Para isso, é proposto e implementado um conjunto de modificações aos algoritmos iniciais, ainda continuando a utilizar o PBFT sem modificações. De seguida, são descritas essas modificações.

Na versão original, os componentes nos clientes e nas réplicas referentes ao Byzantium encontravam-se escritos em Java. No entanto, a versão da biblioteca de replicação bizantina sobre a qual o sistema assentava estava escrita em C, o que levava à necessidade de efectuar várias transições de contexto entre o código C e Java. Este facto penalizava o desempenho e tornava complexa qualquer alteração ao sistema, porque existiam partes do algoritmo a executar no código Java e outras no código C. Para evitar esta complexidade, a segunda versão do Byzantium foi implementada praticamente de novo, desta feita completamente em Java (excepto o sistema de base de dados).

A versão original do sistema Byzantium possui alguma margem para optimizações que foi explorada neste trabalho. Primeiro, a propagação das operações no momento do *commit* impõe um peso desnecessário a esta operação, em termos de comunicação. Assim, nesta versão o cliente propaga as operações imediatamente para todas réplicas, através de uma primitiva de comunicação em grupo. Desta forma, no momento do *commit*, todas as réplicas possuem todas as operações que foram executadas durante a transacção. Segundo, na versão original, uma operação de leitura é submetida em todas as réplicas no sistema, quando apenas seria necessário submetê-la em f+1 réplicas correctas do sistema. A colecção de f+1 respostas iguais provenientes de réplicas diferentes permite verificar a correcção do resultado obtido. Terceiro, as operações de leitura numa base de dados SI não bloqueiam outras transacções. Deste modo, é possível executar essas operações sem coordenação nas várias réplicas, fazendo com que a operação de *commit* de uma transacção de leitura termine imediatamente sem troca mensagens, sendo apenas necessário verificar os resultados recebidos durante a execução da transacção. Todas estas melhorias foram incluídas na implementação de forma conjugada, uma vez que abordam fases distintas do algoritmo.

Assim, para cada transacção, existe um conjunto de f+1 réplicas responsáveis por executar imediatamente as operações de leitura da transacção. Cada uma dessas réplicas, após executar uma operação de leitura, envia o resultado para o cliente. Desta forma, no momento do *commit*, o cliente terá já recebido f+1 confirmações do resultado de todas as operações de leitura,

podendo garantir imediatamente o sucesso desta operação. Esta alteração permite reduzir o número de réplicas envolvidas em cada transacção de leitura, reduzindo a carga em cada uma delas e implicitamente abre portas à distribuição de carga pelas réplicas. Adicionalmente, reduz a latência na execução das transacções de leitura que no momento do *commit* podem terminar imediatamente.

Esta aproximação é também aplicada às transacções de escrita. Tipicamente, uma transacção é iniciada por algumas operações de leitura seguida de operações de escrita. Tendo em conta este aspecto, é possível propagar todas as operações de leitura realizadas antes da primeira escrita da transacção para f+1 réplicas no sistema. Assim é possível confirmar o resultado destas operações, reduzindo o processamento necessário para resolver a operação de *commit* é inferior.

1.4 Principais Contribuições

O trabalho apresentado nesta dissertação consiste no desenho e implementação de um sistema de *middleware* para replicação de bases de dados tolerante a falhas bizantinas. Este trabalho apresenta novos algoritmos com as seguintes contribuições principais: suporte à propagação eficiente de operações para as várias réplicas do sistema; utilização de um número reduzido de réplicas na execução das operações de leitura; execução imediata das transacções de leitura sem necessidade de contactar as réplicas, caso não existam erros.

Como se mostra nos resultados apresentados, estas contribuições permitem que o sistema apresente um desempenho semelhante a um sistema não tolerante a falhas com apenas uma réplica, para transacções de leitura. Para transacções de escrita, o sistema apresenta algum *overhead*, como era esperado.

1.5 Organização do Documento

Após este capítulo que faz uma introdução onde se descreve o contexto do problema, a solução proposta e os objectivos a atingir, segue-se o capítulo 2 onde se encontra detalhada a pesquisa efectuada no âmbito deste trabalho apresentando-se algum do trabalho relacionado, nomeadamente sistemas de replicação de bases de dados, sistemas de replicação bizantina e sistemas de replicação bizantina em bases de dados. No capítulo 3, é apresentado o desenho do sistema

implementado, incluindo os modelos considerados, a arquitectura do sistema e o algoritmo de execução de transacções. O capítulo 4 apresenta em algum detalhe a implementação do sistema descrito. Segue-se a descrição dos testes efectuados e dos resultados obtidos no capítulo 5, e terminando o documento no capítulo 6 com as conclusões e o trabalho futuro.

2. Trabalho relacionado

2.1 Replicação em bases de dados

As bases de dados são aplicações informáticas das quais dependem um elevado número de serviços. Sejam aplicações exigentes como as utilizadas por instituições bancárias ou simplesmente para uso pessoal, as bases de dados estão presentes em quase todas as áreas de *software* existentes no nosso quotidiano. Devido à sua vasta utilização, é importante que funcionem correctamente e com um desempenho aceitável. Além disso, é igualmente importante que os dados se encontrem disponíveis sempre que necessário. Um sistema que garanta estas três propriedades perante a ocorrência de falhas aproxima-se do sistema ideal.

Os sistemas de replicação de bases de dados surgem como solução imediata para aumentar a disponibilidade da informação. A ideia de manter cópias completas de uma base de dados em vários locais distintos é interessante para os utilizadores na medida em que reduz relativamente a correlação entre falhas nas diferentes réplicas, e permite o acesso aos dados em caso de falhas em algumas dessas réplicas. Esta solução necessita de lidar com um problema complexo - como manter as réplicas coerentes.

A replicação de bases de dados pode ser classificada em dois tipos: *eager* e *lazy* [12]. A replicação *eager* disponibiliza consistência forte entre as réplicas, normalmente limitando a escalabilidade. A replicação *lazy* atinge melhor desempenho porque permite que as réplicas divirjam, possivelmente expondo os clientes a estados inconsistentes da base de dados. Este trabalho foca-se principalmente na replicação *eager*, uma vez que se pretende abstrair a replicação efectuada e que os clientes tenham a percepção de que existe apenas uma cópia da base de dados. O objectivo é continuar a utilizar as aplicações sem necessidade de efectuar alterações.

As soluções existentes para esse problema podem ser divididas em duas aproximações básicas no que diz respeito à arquitectura. A primeira, primário/secundário (ou *master/slave*) consiste em atribuir a uma das réplicas a responsabilidade de processar as actualizações. As restantes réplicas são geridas pela réplica principal. A segunda, *multi-master* em que todas as réplicas se encontram em igualdade no sistema, sendo possível continuar a efectuar alterações à base de dados mesmo em caso de falha de uma das réplicas. Os sistemas apresentados nesta secção são exemplificativos dessas duas soluções propostas. Nestes sistemas assume-se um modelo assíncrono em que o sistema é constituído por conjuntos de réplicas que trocam mensagens entre si. Assume-se o modelo de falhas *fail-stop* e a não existência de memória partilhada ou

relógios sincronizados.

2.1.1 Generalized Snapshot Isolation

O modelo de isolamento SI foi definido para sistemas com uma só réplica. Do ponto de vista lógico, uma transacção obtém no início da sua execução o último estado da base de dados, onde se encontram reflectidas todas as transacções que efectuaram *commit* antes da transacção iniciar. No momento do *commit* dessa transacção, é necessário verificar se o seu conjunto de escritas se encontra em conflito com os conjuntos de escritas das transacções que efectuaram *commit* durante a sua execução. Se a intersecção dos conjuntos de escrita for não vazia, a transacção T é abortada. Caso contrário, a transacção termina com sucesso.

É de notar que o SI possui duas características de interesse: uma transacção apenas de leitura nunca será bloqueada e uma transacção apenas de leitura nunca fará com que uma transacção de escrita bloqueie ou aborte. A extensão deste conceito a um ambiente distribuído terá consequências sobre estas propriedades. É fácil perceber que a última versão da base de dados pode não estar disponível na réplica acedida. A comunicação necessária para garantir a sincronização do estado da base de dados tornaria as transacções apenas de leitura bloqueantes.

É neste contexto que surge a semântica *Generalized Snapshot Isolation* [9] (GSI). Esta aproximação pode ser descrita como a tentativa de adoptar a abstracção *Snapshot Isolation* (SI) para um ambiente distribuído de base de dados. A ideia principal é baseada na observação de que uma transacção não necessita de ser executada exactamente na versão mais actualizada. É possível executar uma transacção numa versão anterior e ainda assim manter algumas propriedades de SI. Desde que escolhida uma versão adequada, as transacções apenas de leitura não se bloqueiam e não provocam o aborto de transacções de escrita (tal como em SI).

Este relaxamento no critério de escolha da *snapshot* cria um espectro de versões possíveis a utilizar: desde a versão mais recente da base de dados até ao extremo em que todas as transacções executam na versão inicial da mesma. Em [9], os autores propõem a seguinte semântica - *Prefix-Consistent Snapshot Isolation* (PCSI). Num modelo replicado, a posição deste espectro a optar será a última *snapshot* que se encontra disponível localmente e na qual foram executadas as operações anteriores de cada cliente. As propriedades de SI (e de GSI) referidas anteriormente mantêm-se neste modelo. Além disso, este aproximação permite garantir que cada transacção vê reflectida todas as escritas que terminaram com sucesso nessa réplica.

2.1.2 Ganymed

O Ganymed [19] é um sistema de replicação de bases de dados ao nível de *middleware* que oferece consistência e escalabilidade na área das bases de dados replicadas. O sistema opera sem a necessidade de particionar a informação nem de efectuar declarações/extracções dos padrões de acesso das transacções.

O componente central do Ganymed é um escalonador que faz o encaminhamento das transacções para um conjunto de réplicas através do algoritmo de escalonamento RSI-PC (*Replicated Snapshot Isolation with Primary Copy*). Um escalonador RSI-PC é responsável por um conjunto de n réplicas, totalmente replicadas e todas implementando SI. Como se apresenta na figura 2.1, uma das réplicas é usada como *master*, enquanto as restantes n-1 réplicas são *slaves*. O escalonador faz a distinção clara entre transacções apenas de leitura e transacções de escrita, que devem ser assinaladas pela aplicação cliente. Qualquer transacção de escrita é directamente encaminhada para o *master*. O escalonador tem a capacidade de observar a ordem pela qual as transacções fazem *commit*. Após o *commit* bem sucedido de uma transacção de escrita, o resultado é devolvido ao cliente e depois o escalonador assegura que o conjunto das escritas é propagado a todas as réplicas que fazem a instalação das alterações pela ordem que foram executadas no *master*.

Do ponto de vista dos clientes do sistema, o escalonador actua como uma base de dados que disponibiliza SI. Internamente, nem todas as réplicas são actualizadas no âmbito de uma transacção, ou seja, o Ganymed segue uma política *lazy* em relação à actualização das réplicas no sistema. No entanto, disponibiliza um serviço com consistência forte (propriedade da estratégia *eager*), pois o escalonador garante que uma transacção só executa numa réplica actualizada. Assim, o Ganymed reúne as vantagens das estratégias *eager* e *lazy*. Esta aproximação garante as propriedades SI mesmo em ambiente distribuído porque todas as réplicas executam as operações no nível de isolamento SI e mantêm-se coerentes devido à ordenação dos pedidos. Além disso, as leituras são feitas em réplicas actualizadas e todas as operações de escrita são efectuadas na mesma réplica de acordo com a semântica SI implementada pela base de dados.

Caso uma réplica secundária falhe, o escalonador limita-se a ignorá-la no processo de atribuição de transacções até que a mesma tenha sido reparada por um administrador. Se o mesmo acontecer à réplica principal, além de se proceder à eleição de uma nova réplica como *master* existe a necessidade de garantir que não são perdidos os resultados de transacções que já foram concluídas, de modo a assegurar a durabilidade ACID. Isto pode ser conseguido se o resultado

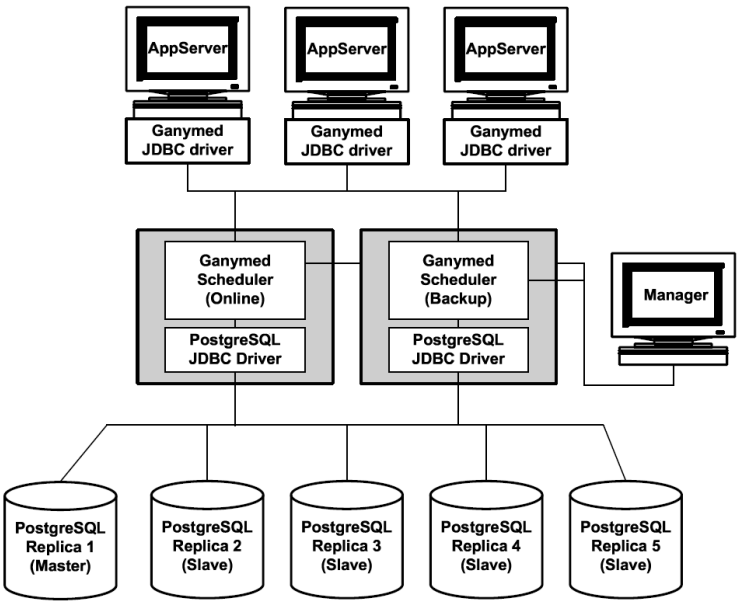

Figura 2.1 Arquitectura do Ganymed (de [19])

da transacção for enviado ao cliente apenas após os conjuntos de escrita terem sido instalados num certo número mínimo de réplicas (a definir pelo utilizador).

2.1.3 GSI - Sistema

Em [9], os autores apresentam um sistema de *middleware* que fornece a semântica GSI. O sistema é composto por um conjunto de réplicas que replicam a base de dados. Os autores propõem dois algoritmos que garantem PCSI: um com uma entidade central de certificação que certifica as transacções de actualização, e outro que replica a entidade de certificação usando uma aproximação de máquina de estados. Em ambos os algoritmos todas as transacções executam as suas operações numa só réplica, excepto a operação de *commit* de uma transacção de escrita que necessita de comunicar para efectuar a certificação. Após o *commit* bem sucedido de uma dessas transacções, o seu conjunto de escritas é instalado nas restantes réplicas.

Na versão centralizada uma das réplicas é considerada o *master*. Essa réplica é responsável pela certificação dos *commits* das transacções de escrita, e guarda os conjuntos de escrita (*writesets*) de todas as transacções bem sucedidas. A certificação das transacções é realizada como em SI. A réplica responsável analisa o conjunto de escritas da transacção que pretende efectuar o *commit* verificando se existem conflitos com os conjuntos de escrita das transacções

que efectuaram alterações bem sucedidas em versões não anteriores à utilizada.

Na versão distribuída o funcionamento é basicamente o mesmo, com a diferença que todas as réplicas executam e certificam as transacções de actualização. Para que isto seja possível, todas as réplicas mantêm a mesma informação que se encontra no *master* da solução centralizada. O funcionamento desta solução depende de um serviço de comunicação em grupo fiável. Este sistema garante que todas as réplicas recebem as operações de *commit* de todas as transacções pela mesma ordem, podendo tomar a mesma decisão de certificação.

2.1.4 Tashkent

Em bases de dados centralizadas, as funções de ordenação dos *commits* das transacções e a acção de tornar os efeitos dessas transacções duráveis são desempenhadas de uma só vez. Por uma questão de eficiência, muitas dessas escritas são agrupadas numa só operação de escrita em disco. Em bases de dados replicadas, nas quais todas as réplicas executam o acordo acerca da ordem de *commit* das transacções de escrita, estas duas funções normalmente encontram-se separadas. Mais especificamente, o *middleware* de replicação determina a ordem global de *commit*, enquanto as réplicas das bases de dados se encarregam de garantir a durabilidade das operações.

O facto de ordenar globalmente as operações e de se depender de um sistema de gestão de bases de dados para tornar as transacções duráveis (ou seja, escrever os seus efeitos em disco) exige que os registos sejam escritos em disco de forma serializada o que constitui um entrave ao desempenho e à escalabilidade do sistema. O Tashkent [24] é um sistema *middleware* que procura melhorias no desempenho em relação aos sistemas tradicionais através da junção destas duas operações (ordenação e escrita). Foram consideradas duas aproximações. Uma das soluções consiste em transportar a durabilidade das transacções para o *middleware*, a outra passa por trazer a informação de ordenação para junto da base de dados.

Para unir a durabilidade com a ordenação no *middleware* (Tashkent-MW), são desligadas as escritas síncronas que garantem a durabilidade na base de dados. Em vez disso, o *middleware* de replicação guarda *logs* das modificações à base de dados (através de conjuntos de escritas) para garantir a durabilidade. As operações de *commit* continuam a ser serializadas em cada réplica da base de dados, no entanto, tornaram-se operações rápidas em memória. O *middleware* encarrega-se de agrupar os conjuntos de escritas no momento da certificação.

Para unir a ordenação com a durabilidade nas bases de dados (Tashkent-API), a API da base

de dados disponível para o *middleware* é estendida de modo a ser possível especificar a ordem pela qual as transacções de escrita fazem *commit*. Assim, a base de dados pode agrupar várias escritas numa só operação de escrita em disco como é realizado normalmente, mas internamente a base de dados força a ordem de *commit* fornecida.

Quando as transacções no sistema apresentam um elevado número de *updates*, as vantagens de unir a durabilidade com a ordenação tornam-se significativas. Em comparação com um sistema semelhante que não implementa este conceito, o Tashkent-MW e o Tashkent-API revelam um aumento no *throughput* na ordem das cinco e três vezes, respectivamente, reduzindo também os tempos de resposta.

2.1.5 Tashkent+

A replicação de bases de dados num *cluster* de servidores é uma aproximação eficiente em termos de custos para escalar bases de dados. No modelo de replicação tradicional, os pedidos de um cliente são interceptados por um componente responsável pelo equilíbrio da carga segundo uma dada política e que permite abstrair dos clientes a natureza replicada da solução. As políticas utilizadas produzem bons resultados, no entanto, podem introduzir contenção ao nível da memória, o que causa um decréscimo do desempenho.

É este o contexto que motivou o aparecimento do conceito MALB (*memory-aware load balancing*). A ideia base deste equilibrador de carga (que toma em consideração o estado da memória para efectuar as suas decisões) passa por utilizar informação acerca da dimensão e conteúdo dos conjuntos de trabalho das transacções de modo a atribuí-las a réplicas que garantem a sua execução em memória (sem acessos a disco). Os desafios principais deste componente estão relacionados com o modo de fazer as estimativas sobre o tamanho dos conjuntos de trabalho e como utilizar essa informação para favorecer a execução em memória.

O sistema Tashkent+ [8] é um *middleware* de replicação de bases de dados que implementa o conceito MALB, assim como a filtragem de actualizações. Basicamente, o funcionamento do MALB é separado em duas partes: a estimação dos conjuntos de trabalho de modo a conseguir fazer uma discriminação em grupos de transacções com o objectivo de colocar um grupo em memória principal numa dada réplica; e a alocação dinâmica dos recursos. O estado da base de dados é continuamente monitorizado e são efectuadas actualizações nas estimativas com o objectivo de alocar dinamicamente a carga existente aos recursos disponíveis. Informação acerca das tabelas acedidas, assim como o estado do CPU e do I/O permitem efectuar a divisão

em grupos de transacções. No fundo, esta técnica consiste em dirigir as transacções dum dado tipo sempre para as mesmas réplicas.

É também utilizado o conceito de filtragem de actualizações. Devido às características deste sistema (especialização de cada réplica num grupo ou tipo de transacções) é possível descartar as alterações propagadas que não visam as tabelas frequentemente usadas por uma dada réplica.

O sistema garante que cada tipo de transacção deve possuir um número mínimo de réplicas onde pode ser executada e cada tabela deve estar disponível em um número mínimo de réplicas. De notar que esta aproximação revela características de replicação parcial, uma vez que as cópias mais actualizadas da informação não se encontram em todas as réplicas do sistema.

2.1.6 Sistema de replicação parcial

Os sistemas descritos anteriormente utilizam uma aproximação de replicação total. No entanto, a replicação total pode nem sempre ser adequada. Primeiro, pode não ser possível manter toda a informação em cada uma das réplicas (por limitações de *hardware*). Segundo, quando existe localidade nos acessos, não há razão para utilizar replicação total na medida em que existe informação a ser replicada que nunca é acedida. Terceiro, a replicação total possui uma capacidade de escalar limitada uma vez que cada transacção deve ser executada em todas as réplicas (ou pelo menos os efeitos da transacção propagados para todas as réplicas).

Em [21], os autores estendem o modelo de replicação DBSM [10] para suportar replicação parcial num *cluster*. O algoritmo proposto é bastante simples e assume o conceito de ordenação espontânea, isto é, com grande probabilidade, as mensagens enviadas para todos os servidores no *cluster* chegarão aos seus destinos pela mesma ordem, algo que é verificado normalmente em redes locais. Cada réplica executa uma sequência de passos (*steps*). Em cada passo, as réplicas decidem acerca do resultado de uma transacção. Um passo é composto por duas fases, uma fase de consenso e uma fase de votação. A fase de consenso é utilizada para garantir que as réplicas concordam com a ordem pela qual as transacções fazem *commit*. Na fase de votação, as réplicas trocam os resultados dos testes de certificação para garantir que os *commits* efectuados seguem uma execução serializável. As duas fases do protocolo (consenso e votação) encontram-se sobrepostas com o objectivo de eliminar uma ronda na comunicação. Basicamente, após a recepção de um pedido de um cliente, a réplica propõe a ordem dessa operação, e imediatamente a seguir procede à votação do seu resultado. Espera-se que quando a fase de consenso de uma transacção T terminar, as réplicas envolvidas concordem que T seja a próxima

	Ordenação das Operações	Replicação	Submissão das operações de escrita
Ganymed	Centralizada	Total	Servidor Central
GSI Tashkent Tashkent+	Centralizada/Distribuída	Total Total Total/Parcial	Qualquer réplica
SRP	Distribuída (com. em grupo)	Parcial	Qualquer réplica

Tabela 2.1 Síntese dos sistemas apresentados nesta secção e suas características.

operação (devido à ordenação espontânea), e que tenham recebido os votos para T sendo capazes de decidir o seu resultado. Se isso não acontecer, ou seja, a ordem acordada é diferente da esperada, as réplicas adaptam-se à nova ordem enviando o resultado da certificação para a próxima transacção escolhida. Esta aproximação garante que todas as réplicas convergem para o mesmo estado porque as transacções são processadas pela mesma ordem nas réplicas acedidas.

Este algoritmo certifica as transacções sequencialmente. Assim, se muitas transacções forem submetidas, pode formar-se uma cadeia de transacções à espera de efectuar *commit* em cada uma das réplicas. Os autores propõem ainda outra versão do algoritmo que resolve este problema ao permitir que uma sequência de transacções seja proposta em cada passo.

2.1.7 Considerações finais

Os sistemas apresentados anteriormente exemplificam diversas técnicas de replicação que têm como principal objectivo o aumento da disponibilidade da informação replicada. A maioria do trabalho realizado concentra-se em soluções de replicação total, geralmente atribuindo maior importância a uma réplica principal que coordena as restantes (Ganymed, Tashkent). Existem também soluções de replicação parcial [21] que diminuem a dimensão dos conteúdos replicados em cada réplica.

Do ponto de vista da arquitectura, todos os sistemas se encontram ao nível *middleware*, ocultando aos clientes e aos servidores a natureza replicada da solução e abstraindo os pormenores de implementação dos serviços replicados.

A solução a desenvolver neste trabalho também é uma solução de *middleware* e utiliza ideias propostas nestes sistemas como a certificação das transacções em diferentes réplicas, estendendo-as e adaptando-as para lidar com falhas bizantinas.

2.2 Replicação Bizantina

Tradicionalmente, nos sistemas distribuídos é comum assumir que os componentes do sistema apenas podem falhar por omissão (falhas dos tipo *fail-stop*), deixando de funcionar completamente. Esta assunção permite simplificar o desenho do sistema desenvolvido.

No entanto, um sistema distribuído fiável deve ter a capacidade de lidar com falhas bizantinas em um ou mais dos seus componentes. Esses componentes falhados podem apresentar um comportamento arbitrário, nomeadamente, enviar informação incorrecta ou conflituosa para diferentes partes do sistema. O problema de lidar com este tipo de falha foi inicialmente introduzido como o Problema dos Generais Bizantinos [16].

O Problema dos Generais Bizantinos consiste no seguinte: um general, comandante de todo o teatro de operações, pretende enviar uma ordem a todos os outros generais (responsáveis por uma divisão das tropas, encontrando-se suficientemente distantes e sendo a passagem de mensagens o único meio de comunicação). O resultado final deve corresponder a uma das seguintes situações: todos os generais leais devem cumprir a mesma ordem, e se o comandante for leal, todos os generais leais cumprem a ordem que ele enviar.

Este problema representa na perfeição o problema das falhas bizantinas em sistemas distribuídos. O comandante representa a réplica que pretende iniciar uma operação distribuída e chegar a um consenso, e os generais representam as réplicas no sistema. A característica de ser leal está relacionada com o correcto funcionamento de uma réplica enquanto um traidor representa uma réplica que apresenta um comportamento arbitrário.

Note-se que, uma vez que não são usadas mensagens assinadas, se provou que a solução deste problema para três intervenientes é impossível. Vejamos as seguintes duas situações, ambas com três participantes. Na primeira, um general traidor envia diferentes ordens aos outros dois. Os dois generais leais trocam valores entre si, ficando com exactamente o mesmo estado. Na segunda situação, um general leal envia a mesma ordem aos outros dois, no entanto, um deles é traidor e na fase da troca da informação envia algo diferente.

Nestes dois casos, o general que é correcto vê exactamente o mesmo cenário: a ordem recebida directamente do comandante é diferente da ordem recebida indirectamente do outro general. Assim, é impossível resolver o problema porque se o general assumir que o comandante é leal deve decidir o seu valor mas se não for deve decidir um valor pré-definido de forma a que os dois generais leais decidam o mesmo valor (note-se que ambos receberam valores diferentes do comandante, pelo que não podem decidir pelo valor recebido).

Este conceito é generalizável para qualquer dimensão de sistema concluindo-se que não é possível lidar com f réplicas falhadas bizantinamente num sistema com dimensão inferior a 3f+1.

Como se mostrou nas situações anteriores, é a capacidade de mentir por parte dos traidores que torna o problema tão difícil. Se essa capacidade puder ser restringida, a solução torna-se mais simples. Uma das formas de o resolver é forçar os participantes a enviar mensagens com assinaturas impossíveis de forjar e cuja autenticidade seja verificável por qualquer um. Com esta adição, o argumento anterior de serem necessários quatro participantes para lidar com um traidor perde-se. Na verdade, existe uma solução com apenas três participantes. O envio de mensagens assinadas pelo comandante na primeira fase impossibilita a tentativa de comprometer o resultado final por parte de um dos generais, uma vez que não têm a capacidade de forjar a ordem recebida. Por outro lado, o facto de as mensagens assinadas serem trocadas por todos os participantes permite-lhes detectar com certeza uma situação de comandante bizantino (porque todos os participantes terão acesso a todas as mensagens recebidas por generais leais).

Este problema introduziu um modelo de sistema bastante poderoso. O modelo de falhas bizantinas é um modelo de sistema que assume basicamente todas as falhas possíveis. Pode ser usado para tolerar falhas de *software*, *hardware* e computadores controlados por atacantes. Em geral, assume-se um modelo de sistema assíncrono onde os nós se encontram ligados por uma rede. Não existem garantias dadas pela rede, logo esta pode perder, atrasar, duplicar e trocar a ordem às mensagens. Quanto aos nós no sistema, estes podem possuir um comportamento arbitrário, ou seja, enviar ou recusar o envio de qualquer mensagem, podem parar a qualquer instante e podem perder ou alterar informação. Também se assume a existência de um atacante com capacidade de coordenar réplicas falhadas com o objectivo de comprometer o sistema. No entanto, existem duas salvaguardas feitas: primeiro, assume-se que o atacante não possui poder computacional para quebrar em tempo útil as técnicas criptográficas utilizadas; segundo, existe um máximo de réplicas falhadas em simultâneo.

Nas secções seguintes apresentam-se um conjunto de sistemas de tolerância a falhas bizantinas.

2.2.1 Pratical Byzantine Fault Tolerance

O PBFT [5] é algoritmo de replicação que tem a capacidade de tolerar a ocorrência de falhas bizantinas. Este algoritmo pode ser usado por qualquer serviço determinista replicado apresentando uma resiliência óptima com pelo menos 3f+1 réplicas para lidar com f réplicas falhas.

Cada réplica mantém o estado do serviço e tem a capacidade de executar todas as operações. As réplicas evoluem no tempo atravessando uma sucessão de vistas. Em cada vista, uma das réplicas é a réplica primária e as restantes são réplicas secundárias. O processo de mudança de vista é iniciado sempre que é detectada uma falha na réplica principal. O funcionamento normal do algoritmo é basicamente o seguinte: o cliente envia o pedido à réplica primária; a réplica primária propaga o pedido para as réplicas secundárias; todas as réplicas no sistema processam o pedido e enviam o resultado ao cliente; o cliente aguarda por f+1 respostas de diferentes réplicas com o mesmo resultado - esse é o resultado da operação.

Quando a réplica primária, p, recebe um pedido, m, é iniciado um protocolo de três fases para atomicamente enviar o pedido para as réplicas, como está representado na figura 2.2. As três fases são: *pre-prepare*, *prepare* e *commit*.

A réplica primária atribui um número de ordem, n, ao pedido e difunde para todas as outras réplicas uma mensagem de *pre-prepare* contendo m. Após a aceitação desta mensagem, uma réplica secundária entra na fase *prepare* ao difundir uma mensagem *prepare* (com n) para todas as outras réplicas.

Uma réplica diz-se no estado preparado se contiver no seu *log* o pedido m, uma mensagem de *pre-prepare* para m e 2f mensagens *prepare* de diferentes réplicas. Nesta situação, a réplica tem a certeza da validade do valor que possui uma vez que esse facto foi assinado por 2f réplicas distintas. Assim que atinge o estado preparado, a réplica difunde uma mensagem de *commit*. Isto inicia a terceira e última fase do protocolo. A operação é executada numa réplica se a mesma se encontrar no estado preparado e tiver recebido 2f+1 mensagens de *commit* referentes ao *pre-prepare* do pedido m, o que garante que 2f+1 réplicas concordaram em avançar com a execução. Após a execução da operação, o resultado é enviado directamente ao cliente.

O sistema incluiu um protocolo de mudança de vista que garante a propriedade *liveness* ao permitir que o sistema evolua mesmo em caso de falha na réplica primária. Este processo é iniciado quando se suspeita da falha da réplica primária, e garante a execução das operações

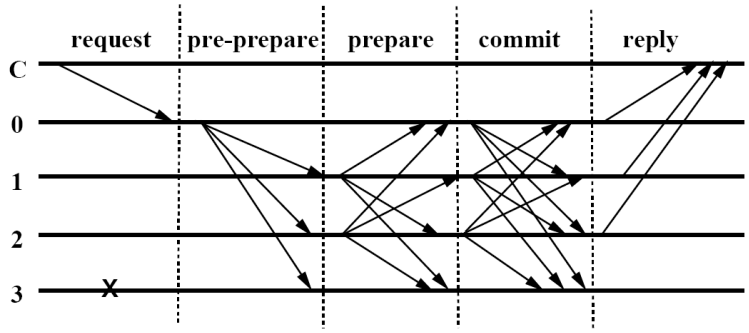

Figura 2.2 Fases do protocolo PBFT (de [5])

2.2.2 BASE

O PBFT fornece tolerância a falhas bizantinas mas necessita que todas as réplicas executem a mesma implementação do serviço e que realizem as alterações ao seu estado de forma determinista. Assim, não possui a capacidade de detectar erros de *software* deterministas que causam falhas em todas as réplicas e além disso, é bastante difícil a reutilização de implementações já existentes devido às extensas modificações necessárias para garantir valores idênticos no estado de cada réplica.

O BASE estende o PBFT definindo uma metodologia que permite a execução de implementações diferentes e não deterministas de um serviço nas réplicas. Esta metodologia divide-se em três partes. Primeiro, é necessário definir uma especificação abstracta para o serviço, onde é especificado o estado abstracto, o seu valor inicial e o comportamento de todas as operações que o manipulam. Em seguida são criados *wrappers* de conformidade para cada uma das implementações distintas de modo a fazer com que se comportem de acordo com a especificação comum. Por fim, é implementada uma função de abstracção (e uma das suas inversas) para fazer o mapeamento entre o estado concreto de cada implementação e o estado abstracto comum. Estas funções permitem a transferência do estado entre réplicas, e têm utilidade na reparação de réplicas falhadas ou actualização de réplicas mais lentas. O funcionamento do sistema está dependente do acordo realizado pelas réplicas sobre o valor do estado do serviço após a execução de uma operação. Este acordo não pode ser efectuado sobre o estado concreto, no entanto, a metodologia usada permite a realização do acordo sobre o estado abstracto de cada réplica.

Apesar do uso de abstracção para ocultar os pormenores de implementação, alguns serviços podem conter informação não determinista que não pode ser escondida. Por exemplo, se o

tempo em que um ficheiro foi acedido pela última vez num serviço NFS estiver configurado para ser lido do relógio local do servidor. Caso essa execução seja independente, o estado das réplicas pode divergir. A solução para este problema passa por permitir que a réplica primária proponha valores para operações desta natureza. Se a réplica primária estiver falhada, o envio de valores diferentes é detectado aquando do acordo efectuado entre as restantes réplicas, no entanto, se for enviado o mesmo valor errado, o comportamento do sistema pode não ser o desejado. Esta situação é resolvida através da implementação, em cada réplica, de uma função de validação para valores não deterministas propostos. Se um terço ou mais réplicas rejeitarem o valor, é iniciado o processo de mudança de vista que, quando terminado, deixará uma nova réplica como réplica primária.

2.2.3 Zyzzyva

O Zyzzyva [14] é um sistema de replicação tolerante a falhas bizantinas. O algoritmo implementado pelo sistema explora a execução especulativa nos servidores como forma de reduzir o custo e simplificar o desenho do sistema (em comparação com o sistema PBFT). Assim, as réplicas vão executar as operações sem terem a certeza da ordenação das operações. O protocolo é executado em 3f+1 réplicas (resistente a f réplicas bizantinas) e a sua execução é organizada numa sequência de vistas. No contexto de uma vista, uma das réplicas é designada como primária, responsável por coordenar o sub-protocolo de consenso.

O protocolo de consenso é iniciado quando um cliente envia um pedido à réplica primária. Ao receber uma mensagem com o pedido, a réplica primária atribui um número de sequência à mesma e propaga o pedido para as outras réplicas. Cada réplica executa especulativamente o pedido e responde ao cliente enviando uma mensagem com o resultado. Nesta fase do protocolo resta ao cliente aguardar pelas respostas, coleccionando as mensagens correspondentes ao pedido efectuado. A próxima acção a executar está dependente dos resultados recolhidos. Se o cliente receber 3f+1 respostas iguais, pode concluir que mesmo na presença de f réplicas bizantinas, o sistema poderá continuar a evoluir. Nesta situação, entrega a resposta à aplicação e completa o pedido. Se o cliente receber entre 2f+1 e 3f mensagens de resposta, é criado um certificado de *commit* que é enviado a todas as réplicas. Quando uma réplica recebe o certificado, confirma a sua recepção enviando ao cliente uma mensagem. Assim que o cliente recebe 2f+1 mensagens destas, entrega a resposta à aplicação e termina o pedido. Este procedimento garante ao cliente que uma parte suficiente das réplicas (pelo menos 2f+1 delas) executou o

pedido correctamente. Se o cliente receber menos de 2f+1 respostas, o pedido é enviado novamente para todas as réplicas que o redireccionam para a réplica primária para garantir que o pedido é numerado e mais tarde executado. Este procedimento é efectuado porque existe a possibilidade de a réplica primária ser bizantina e ter enviado pedidos diferentes às diferentes réplicas.

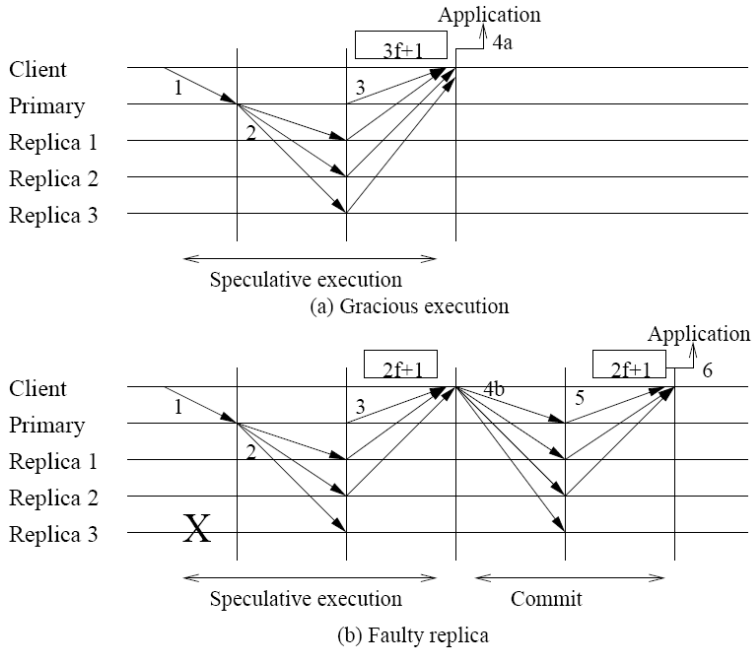

Figura 2.3 Execução do protocolo no caso esperado(a) e em caso de réplicas falhadas(b) (de [14])

Além do protocolo de consenso, o sistema Zyzzyva é composto também por um protocolo de *checkpoint* e um protocolo de mudança de vista. O protocolo de *checkpoint* permite descartar as mensagens obsoletas de modo a limitar a história da execução. Tal como no sistema PBFT, o protocolo de mudança de vista garante a evolução do sistema em caso de existência de falhas bizantinas na réplica primária.

2.2.4 Query-Update

O protocolo Query/Update (Q/U) [2] é uma alternativa às aproximações baseadas em acordo, que são soluções não escaláveis perante falhas. O protocolo Q/U é um protocolo eficiente, optimista e baseado em quórum que permite a construção de serviços tolerantes a falhas bizantinas. Um quórum é um subconjunto de réplicas no sistema. Os objectos que usam o sistema

devem disponibilizar interfaces compostas por métodos deterministas de dois tipos: as *queries* que nunca alteram os objectos e os *updates* que os alteram. O protocolo garante que todas as operações são serializadas. O protocolo opera correctamente no modelo de falhas bizantinas tolerando clientes e servidores com comportamento arbitrário. São necessários 5f+1 servidores para tolerar f servidores falhados. Neste sistema um quórum é formado 4f+1 réplicas do sistema. O número de réplicas contactadas permite garantir a evolução do sistema perante a ocorrência de falhas.

O protocolo Q/U baseia-se em servidores de versões para implementar o seu optimismo. Os objectos Q/U encontram-se replicados em todos os servidores do sistema. Cada servidor mantém as versões dos objectos associadas a uma estampilha lógica formando o que se chama a história da réplica. Para executar uma operação, um cliente primeiro recebe a história das réplicas e guarda-o no seu conjunto de histórias de objectos (OHS - *object history set*), que se resume a um *array* de histórias de réplicas indexado pelo número do servidor. Depois de devidamente actualizado, o cliente coloca o OHS, o método a invocar e os seus argumentos num pedido e envia-o aos servidores. Ao enviar o seu OHS, o cliente comunica ao servidor informação relativa ao estado global do sistema. Cada operação de update invocada pelo servidor de versão resulta na criação de uma nova versão do objecto naquele servidor.

No lado do servidor, após recebido um pedido, é verificada a integridade do OHS associado e é executada a operação. As operações de *update* resultam na criação de uma nova versão do objecto replicado. A estampilha temporal para a nova versão é calculada deterministicamente a partir do OHS e da operação efectuada. Este mecanismo permite evitar a divergência no estado das réplicas. No caso esperado, ou seja sem contenção nem erros, as operações são executadas em apenas uma fase. O cliente entrega o resultado à aplicação após receber 4f+1 respostas de diferentes réplicas.

Quando dois ou mais clientes acedem simultaneamente às réplicas, pode dar-se o caso de nenhum deles conseguir completar um quórum. Esta situação é resolvida através de *backoff* exponencial [1] semelhante à técnica utilizada para resolver as colisões em canais de multi-acesso.

Este aproximação permite que as réplicas evoluam para estados incorrectos (porque diferentes réplicas aceitam diferentes operações sem que sejam formados quórums). Como as réplicas nunca comunicam entre si, cabe aos clientes resolver eventuais divergências no estado do objecto replicado. Quando um cliente detecta que uma réplica se encontra desactualizada, isto é,

ainda não executou as operações mais recentes, executa a operação *inline* que consiste em repetir a execução das operações em falta nessa réplica. Esta operação de reparação apenas pode ser executada se não existir contenção. Se forem detectadas inconsistências e existir contenção no acesso, o cliente procede à reparação das réplicas através de um processo de duas fases - *barrier* e *copy*. A operação *barrier* permite garantir o acesso exclusivo a um quórum de réplicas. Cada réplica apenas aceita um pedido de *barrier* de cada vez. Quando o quórum é reunido, executa-se a operação *copy* que consiste em actualizar o estado de todas as réplicas para a versão mais recente.

É de notar que este sistema tem dificuldade em lidar com situações de muita contenção. Os acessos concorrentes originam operações de reparação que bloqueiam o acesso às réplicas.

2.2.5 Hybrid Quorum Protocol

O objectivo do Hybrid Quorum (HQ) [6] é combinar as vantagens de duas aproximações para efectuar replicação: a aproximação baseada em réplicas secundárias (como o PBFT) e a aproximação baseada em quórum (como o Q/U). No HQ pretende-se reduzir o número de réplicas necessárias para formar quórums (5f+1 no Q/U) e aumentar a escalibilidade do sistema, limitada principalmente pelas necessidades de comunicação de todos para todos (PBFT).

O HQ necessita apenas de 3f+1 réplicas e combina as técnicas de quórum (a dimensão de um quórum é 2f+1) e de acordo para garantir desempenho escalável à medida que f aumenta. Caso não exista contenção, o HQ utiliza um protocolo leve de quórum bizantino no qual as leituras necessitam de uma ronda de comunicação entre cliente e réplicas, e as escritas necessitam de duas rondas. Se ocorrer contenção, é usado o algoritmo de replicação BFT para ordenar os pedidos concorrentes de modo eficiente.

Como referido anteriormente, as operações de escrita são executadas em duas fases (escrita-1 e escrita-2). Quando o cliente envia uma mensagem escrita-1 às réplicas pode receber de cada réplica, uma das seguintes três respostas: uma confirmação indicando que pode prosseguir com o pedido de escrita; uma rejeição da escrita (significa que a escrita foi concedida a outro cliente) ou uma resposta com o resultado da operação. Após receber as respostas das réplicas o cliente age de acordo. As suas opções são: (1) após receber um quórum de confirmações com a mesma estampilha, é formado o certificado correspondente e prossegue para a segunda ronda da escrita - esta situação acontece quando não existe contenção; (2) se receber um quórum de rejeições para a mesma estampilha temporal significa que existe outro cliente que pode iniciar a segunda

fase. Para facilitar o progresso na presença de clientes lentos ou falhados é enviada uma mensagem com o quórum de mensagens reunidas a todos os clientes com. Isto é possível porque quando um cliente é rejeitado, recebe a confirmação do cliente que de facto foi aceite. Por isso, qualquer cliente tem a capacidade de formar o certificado e prosseguir com a execução; (3) se receber confirmações com diferentes estampilhas é necessário enviar a mesma mensagem que no caso anterior (mas neste caso enviando o último certificado conhecido). Esta situação acontece quando uma das réplicas ainda não está a par de uma escrita recente. A mensagem é enviada às réplicas desactualizadas; (4) se for recebida a resposta ao pedido corresponde, estamos na situação em que outro cliente juntou o certificado e avançou com a operação (situação 2); (5) se receber confirmações para operações distintas com a mesma estampilha, há a possibilidade de existir numa situação de contenção e é enviada uma mensagem a indicar este facto às réplicas. A solução deste problema passa por ordenar as operações através do protocolo BFT, que também se encontra a executar nas réplicas. Uma delas actua como a réplica primária do BFT. Quando é recebida a mensagem indicadora de contenção, cada réplica envia para a réplica primária uma mensagem a requisitar o início da resolução. Após receber um quórum destas mensagens, a réplica primária cria uma operação BFT, que executará normalmente, para resolver o conflito. Do ponto de vista dos clientes, apenas existem diferenças ao nível dos certificados recebidos, isto é, todo o processo de resolução de contenção é abstraído.

Quando o cliente reúne condições para entrar na segunda fase, envia o certificado reunido e confirma junto das réplicas a operação a realizar. Depois aguarda por um quórum de mensagens para confirmar o resultado recebido.

As operações de leituras são realizadas numa só ronda de mensagens. O pedido é enviado às réplicas de onde podem surgir dois resultados. Se todas as respostas forem coerentes o resultado é entregue à aplicação. Caso contrário, estamos na mesma situação que o caso (3) da operação de escrita; isto é, existem réplicas que ainda não viram as escritas mais recentes. Para iniciar o seu processo de recuperação, o cliente envia-lhes uma mensagem a informar desse facto.

2.2.6 Separação entre acordo e execução

Em [24], os autores propõem uma solução para diminuir o número de réplicas que são necessárias para tolerar falhas bizantinas. Esta solução distingue-se pelo facto de fazer uma separação clara entre o conceito de acordo em relação à ordem dos pedidos e o conceito de execução desses pedidos. Assim, o sistema requer 3f+1 réplicas de acordo para tolerar f falhas bizantinas em

réplicas de acordo, mas apenas 2g+1 réplicas de execução para tolerar g falhas bizantinas em réplicas de execução (que mantêm o estado do serviço replicado).

A separação do acordo e da execução é possível porque a fase de acordo dos tradicionais algoritmos de replicação bizantina produz uma prova criptograficamente verificável acerca da ordem definida. Este certificado de acordo pode ser verificado por qualquer servidor permitindo assim a separação dos nós de execução dos nós de acordo. Podemos agrupar o algoritmo em três partes: protocolo inter-*cluster*, protocolo interno do *cluster* de acordo e protocolo interno do *cluster* de execução. Não considerando os protocolos internos dos *clusters* de acordo e execução, existem mensagens a transitar entre os clientes e o *cluster* de acordo, e entre o *cluster* de acordo e o *cluster* de execução. De notar que um cliente nunca envia mensagens directamente a uma réplica de execução.

Do ponto de vista de um cliente, este envia um pedido (com a operação pretendida e uma estampilha temporal) certificado por ele próprio para as réplicas de acordo. Após o envio do pedido, o cliente aguarda por uma resposta certificada por pelo menos g+1 réplicas de execução. Se após um *timeout* o cliente não tiver recebido as respostas, o pedido é retransmitido para todas as réplicas de acordo.

Quanto às réplicas do *cluster* de acordo, têm como função ordenar os pedidos, enviá-los para as réplicas de execução e devolver os resultados das execuções aos clientes. Quando o *cluster* de acordo recebe um pedido de um cliente executa três passos, dos quais o primeiro é opcional: verificar se existe em *cache* algum certificado com a resposta para o pedido (mesmo cliente e estampilha temporal igual ou superior); caso exista o certificado é enviado ao cliente e o processamento do pedido termina. Caso contrário, é gerado um certificado autenticado por pelo menos 2f+1 réplicas de acordo de modo a afectar o pedido a um número de ordem, usando o protocolo PBFT. Depois, o certificado de acordo é enviado ao *cluster* de execução. Além disso, as réplicas de acordo ainda necessitam de lidar com as respostas aos pedidos. Ao receber um certificado de uma resposta, o *cluster* de acordo envia-a para o cliente e opcionalmente guarda-o em *cache* (para efectuar a optimização descrita anteriormente). Por sua vez, as réplicas pertencentes ao *cluster* de execução lidam com o estado do serviço e executam os pedidos pela ordem determinada na fase anterior. Para suportar a semântica de execução "apenas uma vez", as réplicas de execução fazem *cache* dos resultados computados. Quando uma réplica do *cluster* recebe um pedido válido e um certificado de acordo válido (com o número de ordem do pedido), espera até que a execução dos pedidos anteriores a esse estejam completados e depois executa a operação.

2.2.7 Tolerância a falhas bizantinas com elevado desempenho

Em [15], os autores propõem um sistema que utiliza informação proveniente da aplicação para identificar e executar concorrentemente os pedidos. Este sistema utiliza a solução proposta no sistema anterior (2.2.6), que permite distinguir entre o processo de acordo sobre a ordem de processamento dos pedidos e a sua execução. A optimização do desempenho é conseguida devido à introdução de um elemento que permite paralelizar a execução das operações independentes. Este elemento realiza a sua operação após a execução do acordo da ordem e imediatamente antes da execução do pedido.

A identificação da relação de independência entre pedidos baseia-se no seguinte conceito: dois pedidos dizem-se dependentes se o conjunto de escritas de um deles tem pelo menos uma variável em comum com os conjuntos de leitura e escrita do outro. Assim, dois pedidos dizem-se independentes se não forem dependentes. Este sistema garante que toda a réplica não falhada no sistema processa qualquer par de pedidos dependentes recebidos sequencialmente e pela mesma ordem relativa.

A limitação principal de um sistema com esta arquitectura deve-se ao facto de as regras usadas pelo "paralelizador"para identificar a independência entre pedidos necessitarem de informação acerca da aplicação. No entanto, nem sempre é necessária esta informação. É possível definir um conjunto de regras conservadoras que apesar de limitarem o paralelismo ainda permitem a execução paralela de uma parte dos pedidos.

Como exemplo desta arquitectura foi implementado um serviço NFS tolerante a falhas bizantinas, o CBASE-FS. Num sistema de ficheiros é possível definir algumas operações independentes. Por exemplo a operação *getattr* é apenas de leitura qualquer que sejam os argumentos. Operações de leitura em ficheiros diferentes são independentes. O mesmo não se passa se forem leituras do mesmo ficheiro uma vez que a operação de leitura modifica a informação da última vez acedida. Do mesmo modo, escritas de ficheiros diferentes são independentes, no entanto, escritas e leitura no mesmo ficheiro são dependentes.

2.2.8 Considerações finais

Nesta secção apresentaram-se algumas das soluções propostas para garantir a segurança da informação perante a ocorrência de falhas bizantinas. O aumento da disponibilidade da informação é conseguido através da coordenação de várias réplicas, executando protocolos de máquina de estados que usam uma réplica primária para ordenar as operações (como o PBFT, o BASE

	Ordenação das Operações	Dimensão	Latência das Operações de escrita
PBFT	Réplica primária	3f+1	5
BASE			5
Zyzzyva			3
Q/U	Quórums	5f+1	2
HQ	Réplica primária + Quórum	3f+1	4
Separating High-Throughput	Ordenação independente das réplicas	2f+1	5 + 2

Tabela 2.2 Síntese dos sistemas apresentados nesta secção e suas características.

ou o Zyzzyva) ou protocolos baseados em quórum como o Query/Update. Existem também as soluções híbridas como é o caso do Hybrid Quorum Protocol que procuram combinar as vantagens das duas aproximações.

Recentemente, demonstrou-se que a separação do acordo e da execução permite a redução do número de réplicas de execução para 2f+1. O sistema que o apresentou pela primeira vez foi descrito em [24], e o sistema [15] tira partido desse resultado melhorando o desempenho através da paralelização de operações independentes.

Na solução a desenvolver neste trabalho usa-se um sistema de replicação bizantina sem modificações. Adicionalmente propõe-se uma solução para mapear as transacções compostas por uma sequência de operações num modelo de replicação de máquina de estados, como o usado pela generalidade dos sistemas aqui apresentados.

2.3 Replicação bizantina em bases de dados

Apesar de existirem vários sistemas de replicação bizantina, poucos são os que foram desenvolvidos especificamente para replicar bases de dados. Nesta secção apresentam-se as iniciativas que foram realizadas nesse sentido.

2.3.1 Replicação diversa

A importância da replicação e da diversidade como forma de tolerância de falhas em sistemas computacionais há muito que é reconhecida. A replicação de *hardware* é normalmente vista

como um método adequado para tolerar falhas arbitrárias de *hardware*. No entanto, falhas em *software* que não provoquem sintomas visíveis como um *crash*, ou que ocorrem em todas as réplicas ao mesmo tempo e em todas as tentativas de execução não serão detectados por sistemas de replicação não diversa.

Este tipo de falhas requer uma solução de replicação diversa que consiste em utilizar implementações distintas da mesma especificação com o objectivo de reduzir a correlação entre os erros no *software*. Apesar de existir há cerca de 30 anos, e ser objecto de estudo constante, esta técnica é pouco utilizada devido aos custos associados [13]. No entanto, o custo de desenvolvimento de versões distintas veio progressivamente a ser eliminado devido ao sucesso de diversos produtos em várias indústrias e ao crescimento do mercado para componentes *off-the-shelf*. A utilização destes componentes constitui uma alternativa de baixo custo para os integradores de sistemas que necessitam de aumentar a fiabilidade.

Em [13], os autores propõem que, nos sistemas de bases de dados, as limitações existentes na replicação não diversa sejam ultrapassadas através da construção de um nó de tolerância a falhas (nó-TF) a partir de dois ou mais servidores SQL, abstraídos numa camada de *middleware* de modo a parecer um servidor junto dos clientes. A conectividade entre cliente e o *middleware* pode ser implementada numa API standard como o JDBC/ODBC. A comunicação entre *middleware* e os servidores pode ser feita com recurso a às soluções de conectividade disponibilizadas pelos servidores escolhidos. A composição deste sistema introduz problemas com os quais é inevitável lidar:

- A sincronização entre os servidores de modo a garantir a consistência da informação entre eles: é necessário ter em consideração aspectos como a ordem de entrega das operações aos servidores e a ordem pela qual os servidores executam as operações que por sua vez são afectadas pelos planos de execução criados e pela execução dos mesmos através dos motores de execução normalmente não deterministas;

- Suporte para tolerância a falhas através dos mecanismos de detecção e contenção de erros e de recuperação de estado: a diversidade introduzida permite a análise dos resultados dos diferentes servidores SQL. Além disso, é possível aumentar a disponibilidade da informação ao permitir a recuperação de um servidor enquanto os restantes atendem os pedidos;

- Determinismo nas réplicas deve ser assegurado uma vez que se está a lidar com implementações distintas. O comportamento de cada servidor pode introduzir inconsistências na base de dados mesmo perante uma execução serializada das mesmas *queries*. Um exemplo desta situação é o diferente funcionamento do mecanismo de geração automática de chaves;

- A diferença da especificação do dialecto SQL suportado pelos diferentes servidores. Existem também as questões das funcionalidades não existentes e também as funcionalidades exclusivas. Além de diferenças em funcionalidades há o problema de diferenças na sintaxe entre diferentes servidores SQL para uma funcionalidade comum;

- Diversidade nos dados: existe potencial para aumentar a tolerância a falhas através da escrita de expressões SQL alternativas logicamente equivalente. As diferentes versões seriam enviadas para os diferentes servidores de modo a aumentar ainda mais a heterogeneidade das operações. De notar que esta técnica faz sentido mesmo no âmbito de um sistema replicado não diverso;

- Efeitos na performance devido à diversidade. Os efeitos podem ser positivos ou negativos conforme o regime utilizado. Intuitivamente se percebe que devido à diversidade existente num sistema, diferentes operações serão concluídas em tempos diferentes. Se o *middleware* aguardar por todas as réplicas sofrerá as consequências dos piores desempenhos de todas as réplicas nos diferentes tipos de operações, no entanto, garante elevada certeza nos valores calculados. Por outro lado, se o *middleware* devolver o resultado da operação assim que a réplica mais rápida responder, existe a possibilidade de diminuir o tempo de resposta devido aos diferentes tempos de resposta das réplicas.

Todos estes aspectos são apontados pelos autores de [13] como relevantes aquando do desenho e implementação de sistemas que pretendam incluir replicação diversa, apesar de não proporem uma solução para parte deles.

2.3.2 Acordo Bizantino aplicado às bases de dados

Em [17], os autores começam por discutir o problema do consenso na presença de falhas bizantinas num sistema síncrono. De seguida, estendem a discussão à aplicação da solução apresentada para efectuar processamento de dados fiável num sistema de computação distribuída. Este trabalho foi o primeiro a propor a solução deste problema em sistemas de bases de dados.

As condições de correcção que se pretende que o sistema satisfaça são as seguintes: os utilizadores devem obter os mesmos resultados que obteriam de um sistema ideal onde não ocorrem falhas; e se uma transacção é correctamente submetida então ela fará parte do escalonamento final. Os utilizadores terão que lidar com o facto de que as transacções não serão executadas sempre. No entanto, o sistema garante que tudo o que é executado é feito correctamente. Outra

propriedade que será interessante incluir é: o tempo de efectuar um *commit* é limitado. A condição força o sistema a decidir sobre a execução ou não execução de uma dada transacção num dado intervalo de tempo. A decisão de efectuar o *commit* é irreversível.

Após a definição dos critérios de correcção é possível discutir as características da solução. A solução passa por replicar a base de dados em vários nós. Como se pretende lidar com f nós bizantinos, serão necessárias 2f+1 réplicas. Desta forma é possível identificar os resultados correctos como sendo os resultados provenientes da maioria das cópias (o número de nós no sistema, n, deve ser igual ou superior a 2f+1).

Também é necessário que todos os nós correctos executem exactamente as mesmas transacções pela mesma ordem. Caso contrário, o estado nos nós correctos poderia divergir e o mecanismo da maioria dos resultados não seria útil (na realidade é possível executar transacções por ordens diferentes desde que os resultados sejam equivalentes). É exactamente neste ponto que o conceito de acordo bizantino é utilizado. Devido às suas características, existe a possibilidade de ordenar totalmente os pedidos recebidos pelas diferentes réplicas de modo a serem executados pela mesma ordem. Esta ordem é utilizada para entregar os pedidos à aplicação que os executará de modo sequencial.

2.3.3 Commit Barrier Scheduling

Em [23], os autores apresentam o sistema HRDB (*Heretogeneous Replicated Database*) para replicação de bases de dados tolerando falhas bizantinas. Este sistema utiliza o protocolo *Commit Barrier Scheduling* (CBS) para garantir a correcção da execução das transacções submetidas concorrentemente por múltiplos clientes. Este sistema explora o facto de apesar de serem necessárias 3f+1 réplicas para a realização de acordo sob o modelo de falhas bizantinas, apenas 2f+1 são necessárias para executar os pedidos.

No sistema HRDB, os clientes não interagem directamente com as réplicas da base de dados. Em vez disso, comunicam com uma entidade central chamada *shepherd*, que actua como *front-end* para as réplicas, coordenando-as.

É assumido que o *shepherd* não é afectado por falhas bizantinas (mas podem ocorrer falhas do tipo *fail-stop*). Os autores indicam que esta assunção é razoável uma vez que a complexidade e quantidade de código existente no *shepherd* é ordens de magnitude inferior à das réplicas.

No CBS, uma das réplicas é designada por primária, e executa as operações ligeiramente antes das restantes réplicas (secundárias). A ordem pela qual as transacções executam o *commit*

na réplica primária determina a ordem de referência utilizada. Cabe ao CBS fazer com que a ordem pela qual as réplicas secundárias executam o *commit* seja equivalente à ordem de referência. Quando o coordenador recebe uma *query* de um cliente, esta é imediatamente enviada ao gestor da réplica primária. A réplica primária executa as transacções utilizando o seu mecanismo de controlo de concorrência interno. Quando uma resposta é enviada ao coordenador, a transacção correspondente é submetida junto dos gestores das réplicas secundárias. Estes gestores mantêm uma colecção de *queries* a ser executadas nas réplicas e procuram submetê-las com o máximo de concorrência possível, para isso respeitando as seguintes regras: (1) uma transacção é processada sequencialmente; (2) os *commits* das transacções são processados pela mesma ordem; (3) se uma *query* de uma transacção T2 foi executada depois do *commit* de uma transacção T1 na réplica primária, ela só é submetida após a réplica secundária fazer *commit* de T1. As respostas são enviadas aos clientes pelo coordenador assim que recebidas da réplica primária. No entanto, no caso do *commit* é necessário aguardar pela resposta de f+1 réplicas com resultados idênticos em todas as *queries* da transacção para garantir que o resultado é o correcto.

Estas regras permitem ao CBS garantir uma execução equivalente a uma execução sequencial nas diferentes réplicas preservando uma grande parte da concorrência existente nos sistemas de gestão de bases de dados. O CBS depende da utilização de *two-phase locking* rigoroso por parte das réplicas. A importância deste mecanismo deve-se ao facto de as transacções necessitarem de todos os *locks* no momento do *commit*. Esta restrição, em conjunto com as regras anteriores, garantem a equivalência das execuções entre as diferentes réplicas.

A ocorrência de uma réplica primária falhada é resolvida com um protocolo de mudança de vista. Este protocolo assegura o progresso do algoritmo. Assim sendo, é necessário salvaguardar o estado de todas as transacções que efectuaram *commit* antes da mudança de vista e eleger uma das réplicas como nova réplica primária. Caso a réplica primária falhe bizantinamente as réplicas secundárias podem não conseguir concluir as suas *queries* e o coordenador actua nessa situação abortando todas as transacções pendentes e elegendo uma das réplicas secundárias que tenha efectuado *commit* a todas as transacções passadas.

2.3.4 Considerações finais

O trabalho realizado enquadra-se nesta categoria. A nova versão do sistema Byzantium (assim como a anterior) permite replicação bizantina de bases de dados, mantendo aberta a possibilidade de introduzir o conceito de replicação diversa. Todos os sistemas estudados representam o estado da arte no que diz respeito a replicação de bases de dados e a tolerância às falhas bizantinas. Ao contrário do HRDB [23], o Byzantium aproveita as características dos sistemas de gestão de bases de dados para implementar um *middleware* de replicação que não utilize um componente central que se assume não poder ser bizantino. Este trabalho é o primeiro a apresentar a solução SI.

3 . Desenho da solução apresentada

O Byzantium é um sistema *middleware* de replicação de bases de dados que fornece o nível de isolamento *Snapshot Isolation*. O sistema tem como objectivo permitir a execução de transacções sobre um conjunto de réplicas, mantendo a coerência das mesmas, mesmo na presença de falhas bizantinas.

Neste capítulo é apresentado o desenho da segunda versão do sistema Byzantium. Inicialmente é descrito o modelo do sistema considerado, assim como o modelo de base de dados. Segue-se a descrição de dois componente utilizados na construção do sistema: o componente responsável pela execução de operações tolerantes a falhas bizantinas e o componente de comunicação FIFO. Finalmente é apresentada a arquitectura do sistema e discutida a correcção do algoritmo implementado.

3.1 Modelo de sistema

O modelo de sistema assumido é baseado no modelo de computação distribuída assíncrono. Os componentes encontram-se ligados por uma rede de comunicação em que as mensagens se podem perder, podem ser corrompidas, atrasadas ou entregues fora de ordem. No entanto, assume-se que a retransmissão sucessiva fará com que uma mensagem seja finalmente entregue, caso o receptor não falhe. Esta condição faz com que o sistema não seja puramente assíncrono, permitindo contornar o resultado da impossibilidade FLP [11].

Neste contexto, assume-se a existência de um conjunto finito de clientes e um conjunto de 3f+1 servidores (ou réplicas) sendo f o número de servidores que podem falhar. Os clientes podem igualmente falhar durante a execução do sistema. Relativamente às falhas dos elementos do sistema, clientes e servidores, assume-se que estas podem ser arbitrárias [16], isto é, além de um elemento parar a sua execução, pode igualmente continuar a executar de forma incorrecta, apresentando um comportamento arbitrário. Assume-se também que apesar de ter a capacidade de coordenar vários elementos falhados, um adversário nunca terá poder computacional para quebrar, em tempo útil, as técnicas criptográficas utilizadas.

3.2 Modelo de base de dados

Uma base de dados é uma colecção de registos logicamente relacionados que se encontram armazenados num sistema computacional. O estado de uma base de dados é modificado através da execução de transacções. No sistema Byzantium, em particular, lida-se apenas com sistemas de gestão de bases de dados que implementam o modelo relacional. Uma transacção começa com uma operação de início de transacção (*begin_trx*), seguida de uma sequência de operações de leitura e operações de escrita, e termina com uma de duas operações: *commit*, para tornar os efeitos da transacção definitivos; ou *rollback*, para cancelar a transacção sem que qualquer operação tenha efeito na base de dados. Uma transacção diz-se apenas de leitura (ou simplesmente de leitura) se não incluir nenhuma operação de escrita. Uma transacção diz-se de leitura e escrita se inclui, pelo menos, uma operação de escrita.

O Byzantium disponibiliza o nível de isolamento *Snapshot Isolation* [4]. Neste nível de isolamento, do ponto de vista lógico, uma transacção executa isoladamente como se existisse uma cópia da base dados exclusiva obtida aquando do início da transacção. A transacção pode terminar com sucesso se não existir nenhum conflito entre escritas com transacções concorrentes, isto é, transacções que terminem depois desta se ter iniciado.

Este nível de isolamento possui características de interesse que variam dependendo da forma como se encontra implementado. As características do nível de isolamento estão dependentes do modo como se encontra implementado o controlo de concorrência. Se o controlo de concorrência seguir uma política de multi-versões optimista, o nível de isolamento *Snapshot Isolation* possui as seguintes duas características: uma transacção de leitura nunca será abortada nem nunca fará com que qualquer outra transacção concorrente seja abortada. Por outro lado, se o sistema de gestão de base de dados utilizar um sistema de controlo de concorrência de multi-versões com *locks* verificam-se as seguintes características: uma transacção de leitura nunca bloqueia nem será bloqueada por qualquer outra transacção.

Se o sistema de gestão de bases de dados usasse um sistema de controlo de concorrência pessimista, baseado em *locks* de leitura e escrita, estas propriedades não se verificam. Quando um recurso é acedido para escrita, este é bloqueado pelo sistema e apenas é libertado quando a transacção terminar. Se o recurso não se encontrar disponível (ou seja, se existe um *lock* incompatível pertencente a outra transacção) a transacção que o pretende fica bloqueada. De modo semelhante, se um recurso é acedido para leitura, o sistema obtém um *lock* de leitura, bloqueando as transacções de escrita seguintes.

O *Snapshot Isolation* é bastante popular, uma vez que bastantes sistemas de gestão de bases de dados o implementam, incluindo o Oracle, PostgreSQL e Microsoft SQL Server. Adicionalmente já foi demonstrado que em determinados contextos e aplicações, incluindo todos os *benchmarks standards* (TPC-A, TPC-B, TPC-C e TPC-W), possui uma execução equivalente à do nível de isolamento estritamente serializado [7]. Finalmente, também foi mostrado como é possível alterar um programa SQL genérico de modo a que execute sob uma base de dados *Snapshot Isolation* de forma equivalente àquela que executaria numa base de dados que implementasse o nível de isolamento estritamente serializado.

3.3 Biblioteca BFT (Algoritmo PBFT)

Para garantir tolerância a falhas bizantinas, o sistema Byzantium recorre a um componente auxiliar: uma biblioteca de replicação bizantina que implementa o algoritmo de replicação PBFT apresentado no capítulo 2. Como referido, essa biblioteca garante a execução correcta de operações na presença de falhas bizantinas.

A biblioteca de replicação bizantina oferece replicação por máquina de estados através de uma interface bastante simples, limitando-se a uma única operação que recebe como argumento a operação que deve ser executada em cada uma das réplicas do conjunto, e que retorna o resultado dessa mesma operação. Em cada uma das réplicas, as operações são executadas sequencialmente. Este componente, além de introduzir no sistema a possibilidade de executar operações tolerantes a falhas bizantinas sobre um conjunto de réplicas, implicitamente garante também a execução segundo uma ordem total dessas mesmas operações. Esta característica terá bastante importância para o protocolo de execução de transacções do Byzantium, como é explicado mais à frente neste documento.

3.4 Módulo de comunicação FIFO

Neste sistema existe também um módulo de comunicação FIFO, suportando comunicação em grupo. Através deste módulo é possível fazer chegar junto de um conjunto de participantes uma mensagem de acordo com a semântica FIFO. Este componente baseia-se no *multicast* convencional, garantindo a fiabilidade da entrega de mensagens caso o emissor não falhe. Esta garantia

da propriedade de fiabilidade assume que as mensagens entre dois participantes correctos eventualmente serão entregues, isto é, a rede pode perder mensagens mas existe um número finito de retransmissões que faz com que uma mensagem chegue ao seu destino.

Este módulo permite igualmente enviar mensagens ponto-a-ponto com as mesmas propriedades. A garantia da ordem apenas se verifica entre mensagens do mesmo tipo.

3.5 Arquitectura do sistema Byzantium

O Byzantium é um sistema de *middleware* que disponibiliza replicação tolerante a falhas bizantinas para sistemas de gestão de bases de dados. A arquitectura do sistema é apresentada na Figura 3.1. Como foi referido anteriormente, o sistema é composto por um conjunto de 3f+1 réplicas e um número finito e variável de clientes. O sistema Byzantium encontra-se desenhado como um conjunto de componentes, que permitem abstrair a natureza distribuída da solução, onde clientes e réplicas comunicam entre si com o objectivo de executar transacções num ambiente de replicação de bases de dados.

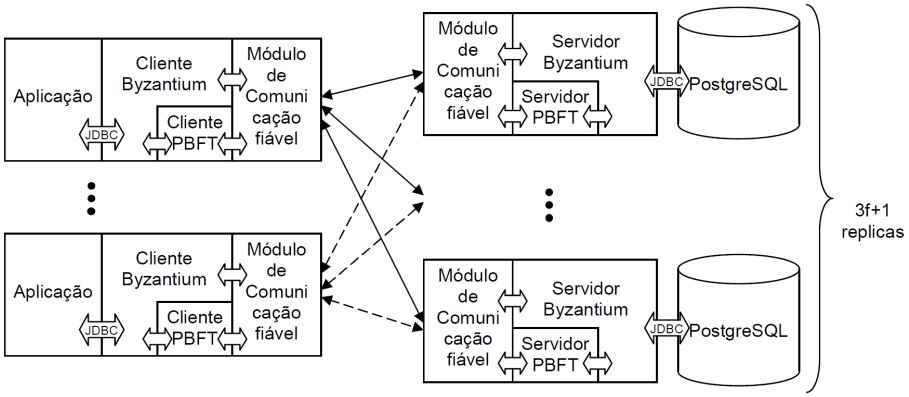

Figura 3.1 Arquitectura da nova versão do sistema Byzantium.

3.5.1 Servidor

Cada servidor é composto por quatro componentes: o componente servidor do Byzantium; o componente servidor da biblioteca do PBFT, um sistema de gestão de base de dados e um módulo de comunicação FIFO.

O componente servidor do Byzantium encontra-se ligado ao componente servidor da biblioteca do PBFT, do qual recebe operações para executar. Essas operações são aquelas que foram sujeitas à serialização através do protocolo de replicação bizantina PBFT. O componente servidor do Byzantium, apesar de receber operações através da biblioteca de replicação, também recebe operações directamente dos clientes Byzantium, através do módulo de comunicação FIFO, sem que estas passem pelo protocolo PBFT. Assim, este componente pode executar concorrentemente várias operações recebidas directamente e uma operação recebida pelo PBFT.

O sistema de gestão de base de dados mantém uma cópia total da base de dados. Os sistemas de gestão de base de dados usados em cada réplica podem ser diferentes, de modo a diminuir a correlação entre possíveis erros de implementação existentes. O único requisito imposto é que disponibilizem uma implementação do nível de isolamento *Snapshot Isolation*.

3.5.2 Cliente

No lado do cliente, as aplicações dos utilizadores dialogam com o sistema utilizando uma interface JDBC oferecida. Desta maneira, aplicações já existentes que sejam compatíveis com a interface JDBC poderão utilizar o sistema Byzantium sem necessitarem de alterações. Para tal, o componente Byzantium no lado do cliente é na realidade uma implementação de um *driver* JDBC.

Este componente encontra-se ligado à biblioteca de replicação BFT, para que tenha a possibilidade de ordenar e executar operações nas réplicas através do protocolo de replicação PBFT. Além disso, também se encontra ligado ao componente de comunicação FIFO, para poder enviar mensagens directamente para todas as réplicas. De seguida, apresenta-se em detalhe o algoritmo de execução de transacções usado no sistema Byzantium. Este algoritmo foi desenvolvido no contexto deste trabalho, sendo uma evolução dos algoritmos originais implementados na primeira versão do sistema [20].

3.6 Algoritmo base

3.6.1 Caso normal

Nesta secção descrevem-se os protocolos envolvidos nas várias fases de execução de uma transacção. Inicialmente é descrito o funcionamento do algoritmo para o caso normal de execução,

sem existência de réplicas ou clientes bizantinos, relegando essa discussão para as secções seguintes. O código executado pelo componente cliente do sistema Byzantium encontra-se na Figura 3.2 e o código executado pelo componente servidor encontra-se na Figura 3.3. Apesar do código apresentar a execução de qualquer tipo de transacções, primeiro será descrita a execução de uma transacção de leitura e escrita, sendo apresentadas as optimizações para lidar com transacções de leitura posteriormente. De modo a manter a apresentação do algoritmo simples foram omitidos os detalhes relativos às situações de erro, tratamento de excepções e assinatura de mensagens. Durante a apresentação do algoritmo, assume-se que o sistema de gestão de base de dados utiliza um sistema de controlo de concorrência optimista, no entanto, a discussão acerca de como lidar com bases de dados com controlo de concorrência baseado em *locks* é apresentada na secção 4.5.1.

A aproximação tomada para maximizar a concorrência e aumentar o desempenho do sistema é limitar o uso do protocolo PBFT às operações necessárias e suficientes que sejam ordenadas totalmente entre si - as operações de *begin_trx* e de *commit*/*rollback*. Esta é a única restrição imposta à execução concorrente de operações de múltiplos clientes. Na solução apresentada, após iniciar uma transacção em todas as réplicas (garantindo a coerência entre os *snapshots*), as operações são executadas especulativamente numa só réplica e a validação dos resultados é atrasada até ao momento do *commit*. Segue-se o detalhe dos algoritmos utilizados para implementar esta aproximação.

A aplicação inicia uma transacção ao executar uma operação *being_trx* (através da função *db_begin*, Figura 3.2, linha 1). O cliente começa por gerar um identificador único para a transacção e seleccionar uma das réplicas para executar a transacção especulativamente - esta será a réplica responsável pela transacção (não é obrigatório ser a mesma réplica primária da biblioteca BFT). O cliente escolhe adicionalmente um conjunto de f réplicas secundárias, usadas para optimizar as transacções de leitura como é detalhado na secção 3.6.6. Depois, o cliente invoca o método *BFT_exex(<begin_trx,...>)* da biblioteca de replicação bizantina que resulta na respectiva chamada em todas as réplicas, ilustrada na figura 3.3, linha 1. Em cada réplica, uma transacção na base de dados é iniciada. Devido às propriedades garantidas pelo sistema PBFT, e como as operações *begin_trx* e *commit* são executadas sequencialmente segundo uma ordem total em todas as réplicas, esta estratégia garante que uma transacção é iniciada numa *snapshot* equivalente da base de dados em todas as réplicas correctas. Inicialmente, todas as transacções assumem-se como sendo transacções só de leitura (*read-only*) até à chegada da primeira operação de escrita, o que permite algumas optimizações que serão discutidas na secção

seguinte.

Após a execução da operação *begin_trx*, uma aplicação encontra-se no estado em que pode executar qualquer sequência de operações de leitura e de escrita (através das funções *db_read_op* e *db_write_op*, Figura 3.2, linhas 12 e 26 respectivamente). No caso de uma operação de escrita, a operação é enviada através do componente de comunicação em grupo para todas as réplicas (através da chamada *mcast*, que resulta na execução da chamada *receive write* no lado das réplicas, que se encontra na Figura 3.3, linha 35). A operação é recebida por todas as réplicas mas apenas é executada pela réplica responsável. O resultado é devolvido ao cliente que por sua vez o devolve à aplicação (Figura 3.2, linha 32). O cliente e a réplica responsável mantêm uma lista contendo a sequência das operações efectuadas e os respectivos resultados.

Por sua vez, as operações de leitura podem ser executadas de duas formas diferentes. Se a transacção onde a operação é executada for uma transacção de escrita, a execução da operação de leitura é semelhante à execução de uma operação de escrita descrita anteriormente. Caso contrário, procede-se à execução da operação de leitura na réplica responsável e réplicas secundárias, como detalhado na secção 3.6.6.

Uma transacção termina quando é executada a operação de *commit* (função *db_commit*, Figura 3.2, Linha 35). Neste momento é necessário confirmar os resultados obtidos em todas as operações que foram executadas apenas numa réplica (potencialmente bizantina). Para isso, o cliente invoca a operação de *commit* através da execução da operação BFT que inclui um *hash* das operações e dos seus resultados. Em cada réplica, o sistema verifica se a execução da transacção é válida antes de efectuar o *commit* na base de dados (este processamento é realizado na chamada no lado das réplicas correspondente a *BFT_exec(<commit,...>)*, Figura 3.3, linha 10).

Para validar uma transacção, executam-se os seguintes passos. Todas as réplica excepto a réplica responsável pela transacção executam as restantes operações da transacção e verificam a sua correcção comparando o *hash* dos resultados obtidos com aquele que o cliente obteve anteriormente (Figura 3.3, linha 14). Após ser garantido que todas as réplicas executam a mesma sequência de operações no mesmo *snapshot*, se a réplica responsável era correcta, então todas as outra réplicas correctas irão produzir a mesma sequência de resultados. Por outro lado, se a réplica responsável era bizantina, os resultados obtidos pelas restantes réplicas não irão corresponder aos que foram recebidos pelo cliente. Neste caso, as réplicas correctas irão abortar a transacção e o cliente assinala o comportamento bizantino através do lançamento de uma excepção.

É ainda necessário garantir que todas as réplicas incluindo a réplica responsável não comprometem as propriedades SI em relação à transacção que pretende terminar. Ao executar na mesma *snapshot*, e efectuar a operação de *commit* pela mesma ordem em todas as réplicas, o processo de *commit* é determinista e provoca o mesmo resultado em todas as réplicas. Assim, como o sistema de base de dados implementa a semântica *Snapshot Isolation*, se uma transacção efectuar *commit* numa réplica correcta, irá terminar com sucesso em todas as réplicas correctas.

Uma transacção também pode terminar através da operação de *rollback*. Uma solução imediata é simplesmente abortar a transacção em todas as réplicas. Na secção 3.6.5 discutem-se os problemas desta aproximação e propõe-se uma alternativa.

3.6.2 Tolerar clientes com comportamento bizantino

O sistema deve tolerar comportamento bizantino por parte dos clientes. Em particular, é importante assegurar que um cliente nesta situação não tem a capacidade de provocar uma divergência no estado das réplicas. Note-se que o objectivo não é prevenir que um utilizador mal intencionado com acesso legítimo à base de dados utilize a interface do sistema para comprometer a informação guardada. Ataques desta natureza apenas podem ser evitados através do uso de políticas de segurança/controlo de acesso restritas assim como a manutenção de várias *snapshots* da base de dados com o propósito de efectuar recuperação de dados, caso seja necessário. O objectivo é prevenir a divergência das réplicas correctas e violações à semântica da base de dados.

Tal como se encontra em [5], o PBFT é usado para executar operações num conjunto de réplicas que serão executadas segundo uma ordem total de forma serializada. Como este algoritmo lida com a existência de clientes bizantinos ao assegurar que os mesmos não podem modificar o estado do sistema excepto através da interface disponibilizada, o sistema Byzantium apenas necessita de lidar com a validade das operações que são submetidas nos sistema de gestão de bases de dados no lado das réplicas.

Primeiro, as réplicas devem verificar a validade da sequência de operações recebida de cada cliente. A maioria das verificações são simples, como confirmar que uma operação *begin_trx* é seguida por uma operação *commit*/*rollback* ou verificar se os identificadores únicos são válidos.

Existe uma situação problemática associada à ocorrência de uma falha bizantina num cliente. No momento do *commit*, o cliente envia apenas um *hash* da lista de operações. Operações essas que são enviadas para todas as réplicas durante o decorrer da transacção pelo cliente.

Como estas operações são enviadas directamente durante a execução da transacção, pode dar-se o caso de haver diferenças entre a sequência de operações em cada réplica. Essas diferenças constituem um problema na medida em que durante a operação de *commit*, as réplicas com a sequência correcta irão terminar com sucesso, e as restantes irão abortar a transacção. Esta divergência no estado das réplicas constitui um problema, cuja solução é discutida em seguida.

Para resolver este problema, tira-se partido do mecanismo existente no protocolo PBFT que permite que as réplicas cheguem a acordo acerca de escolhas não deterministas (originalmente usado para chegar ao consenso acerca de valores como por exemplo o valor do relógio). Aplicando este conceito ao problema em questão, é possível fazer com que as réplicas cheguem a acordo acerca da lista de operações da transacção antes de executar qualquer operação. Neste processo de decisão, cada uma das réplicas representa o seu voto através de um simples boleano, que representa o facto de a réplica concordar com a lista de operações preste a ser executada.

Internamente na biblioteca BFT, a réplica primária propõe esse valor para consenso junto das restantes, e caso 2f+1 no total concordem com o mesmo, a operação de *commit* prossegue normalmente. Esta decisão pode implicar que algumas réplicas correctas que não se encontrem neste conjunto, isto é, que não tenham a sequência de operações correcta necessitem de a obter junto das outras réplicas. Se o valor proposto pela réplica primária for negativo, e se 2f+1 réplicas concordarem, a transacção é abortada, no entanto, se não houver consenso acerca do valor proposto, o processo de mudança de primário no PBFT é iniciado, e a nova réplica primária inicia um processo de decisão onde cada réplica no sistema propõe um valor (desta vez usando a sequência de operações), e cada réplica decide entre os valores propostos após aplicar uma função determinista sobre os mesmo. Assim, desde que exista uma réplica com a sequência de operações correcta, a transacção irá terminar com sucesso.

Além disso, existe ainda a situação em que uma réplica responsável correcta necessita de descartar as alterações efectuadas durante a execução de uma transacção. Isto deve-se à possibilidade de um cliente bizantino enviar, no momento do *commit*, uma lista de operações diferente da que executou previamente. Para lidar com esta situação recorre-se ao popular mecanismo de *savepoints* que permite efectuar *rollback* para um estado anterior dentro de uma transacção, descartando todas as alterações efectuadas entretanto. Para tal, no início de uma transacção é criado um *savepoint*. Se durante o processo de *commit* é necessário executar uma sequência diferente de operações, em consequência do passo anterior, a réplica retorna ao *savepoint* criado e executa o processo normalmente usando a nova sequência de operações. Este procedimento é necessário para evitar que a réplica responsável evolua para um estado diferente das restantes

réplicas, ao garantir que todas as réplicas correctas, incluindo a réplica responsável, executam a mesma sequência de operações sobre o mesmo *snapshot* da base de dados.

Por fim, é necessário garantir os resultados devolvidos inicialmente pela réplica responsável corresponde aos resultados indicados pelo cliente na operações de *commit*. De modo a prevenir que um cliente provoque uma divergência no estado das réplicas, é necessário verificar na réplica responsável no momento do *commit* se o *hash* dos resultados corresponde aos obtidos durante a execução. Se os valores coincidirem, a operação termina com sucesso, caso contrário a transacção é abortada. Esta simples verificação evita que a réplica responsável termine correctamente uma transacção que todas as restantes réplicas no sistema irão abortar.

3.6.3 Tolerar uma réplica responsável com comportamento bizantino

A réplica responsável por uma transacção pode falhar, devolvendo resultados errados ao cliente ou não devolvendo resultado algum. Caso ocorra a primeira situação, a aplicação prossegue a sua execução normal até ao momento do *commit*. De notar que estes resultados não são verificados e a aplicação prossegue com base em resultados possivelmente incorrectos. No entanto, quando a operação de *commit* é realizada, cada réplica compara os seus resultados com os enviados pela réplica primária para o cliente, comparando os *hashes*. Se for verificada alguma discrepância nos valores, as réplicas correctas assinalam esse facto devolvendo um resultado negativo à operação de *commit*, e a transacção é abortada. Assim, pelo menos 2f+1 réplicas correctas concordam em abortar a transacção, o que garante a correcção e a consistência no sistema.

Se o cliente não conseguir obter nenhum resultado da réplica responsável, selecciona uma nova réplica responsável e repete as operações executadas durante essa transacção até esse ponto. Se os resultados não coincidirem, o cliente termina a transacção com um *rollback* e lança uma excepção indicando comportamento bizantino. Caso contrário, a transacção pode continuar normalmente. No momento do *commit*, uma réplica que assuma ainda ser a réplica responsável pela transacção continua a verificar se a sequência de operações enviada pela cliente está de acordo com as operações executadas na réplica até ao momento da mudança. Assim, se a réplica responsável que foi substituída ainda se encontra activa, irá identificar que operações adicionais foram executadas. Nesse momento, as operações iniciais são descartadas e após obter a nova lista de operações, executa-a tal como qualquer outra réplica. Este mecanismo

assegura um funcionamento correcto do sistema, uma vez que todas as réplicas, incluindo réplicas responsáveis substituídas, executam a mesma sequência de operações na mesma *snapshot* da base de dados.

3.6.4 Tolerar uma réplica não responsável com comportamento bizantino

Quando uma réplica não responsável tem comportamento bizantino, isso não afecta a execução normal do protocolo apresentado inicialmente. Este facto deve-se a que esta réplica apenas intervém em operações executadas pelo PBFT [5], o qual já lida com a situação de uma réplica ser bizantina.

3.6.5 Lidar com transacções abortadas

Quando uma transacção termina com a operação *rollback*, uma aproximação possível consiste em simplesmente abortar a transacção em todas as réplicas no sistema, sem efectuar qualquer verificação aos resultados devolvidos (como a solução adoptada no HRDB [23]). No sistema Byzantium isso pode ser facilmente implementado através da execução de uma operação que aborte as transacções nas réplicas, submetida através da biblioteca de replicação BFT. Esta aproximação não representa nenhum problema na consistência entre as réplicas uma vez que a base de dados nunca chega a ser alterada. No entanto, caso a réplica responsável pela transacção apresente comportamento bizantino, a aplicação pode ter observado um estado incorrecto da base de dados, o que pode dar origem a uma operação de *rollback* desnecessária.

Para detectar esta situação, tomou-se a decisão de incluir um mecanismo opcional para verificar a correcção dos resultados obtidos durante uma transacção que termina com *rollback*. Com esta opção activada, a execução de um *rollback* torna-se semelhante a uma operação de *commit*, apenas com a diferença que a transacção irá abortar sempre. Se a verificação dos resultados falhar, a operação de *rollback* lança uma excepção assinalando comportamento bizantino. Note-se que uma aplicação correcta deve apanhar todas as excepções lançadas durante as operações executadas sobre a base de dados e lidar com essas ocorrências.

3.6.6 Optimizações às transacções de leitura

Num sistema tolerante a falhas bizantinas, é possível verificar a correcção da execução de uma operação de leitura, caso se recebam f+1 respostas iguais [5]. Esta propriedade pode ser usada

para optimizar o processamento das operações de leitura numa transacção e em particular, o processamento completo das transacções de leitura.

Quando uma transacção é iniciada, assume-se que a mesma é apenas de leitura, até ao momento em que ocorre a primeira operação de escrita. Esta decisão permite identificar o tipo da transacção sem que seja necessário efectuar uma declaração prévia por parte da aplicação. Isto é importante na medida em que permite efectuar optimizações a transacções deste tipo de modo transparente. Além disso, permite ainda fazer o mesmo tipo de optimizações às operações de leitura iniciais das transacções de escrita. De seguida apresentam-se as optimizações efectuadas, as quais se encontram representadas nos algoritmos das Figuras 3.2 e 3.3.

As operações de leitura são executadas imediatamente em f+1 das réplicas (escolhidas pelo cliente) sendo a primeira das f+1 respostas devolvida à aplicação, e as restantes f respostas recolhidas em *background*. Como o modelo de falhas considerado é o modelo de falhas bizantinas, onde se assume que no máximo f réplicas podem falhar simultaneamente, é seguro afirmar que f+1 resultados iguais, provenientes de réplicas diferentes, são suficientes para garantir a correcção do valor obtido. Por outro lado, como as operações de leitura não modificam o estado da base de dados, não é necessário executá-las em todas as réplicas. Assim, o cliente mantém informação sobre quais as operações que se encontram confirmadas.

No momento do *commit*, se os resultados de todas as operações forem validados deste modo, o cliente pode retornar imediatamente e a transacção termina com sucesso. Caso isto não se verifique é necessário contactar réplicas adicionais e para isso, recorre-se à execução da operação de *commit* base, a mesma usada para transacções de escrita. Neste caso é possível verificar apenas o sufixo de operações após a última operação confirmada.

Esta alteração permite reduzir a carga em cada uma das réplicas assim como introduzir no sistema a possibilidade de execução paralela real de transacções, uma vez que as transacções só com leituras apenas executam em f+1 réplicas. Efectuando uma escolha cuidada de quais as réplicas que tratam desse tipo de transacções, é possível equilibrar a carga entre as réplicas do sistema. Diferentes soluções podem ser adoptadas para resolver este problema (por exemplo, como a utilizada no Tashkent+ [8]). No sistema apresentado, é usada uma escolha aleatória.

De notar que esta optimização é possível para operações de leitura mesmo em sistemas de base de dados multi-versão com *locks* porque as operações de leitura nunca bloqueiam. No entanto, o mesmo não se verifica quando se trata de operações de escrita. Particularmente em sistemas de gestão de bases de dados com controlo de concorrência baseado em *locks*, a existência de operações de escrita concorrentes originará a ocorrência de bloqueios. Para

garantir uma execução global sem bloqueios, seria necessário garantir que um dado *lock* fosse ganho pela mesma transacção em todas as réplicas (como no HRDB [23]). Para tornar isso possível ao nível de *middleware* seria necessário definir um conjunto de restrições às operações de modo a ordenar a sua execução. Além da complexidade introduzida, essa solução iria impor atrasos na execução das operações.

3.6.7 Correcção do algoritmo

Nesta secção apresenta-se as condições de correcção para o sistema e descreve-se, de forma informal, o cumprimento dessas condições por parte da implementação realizada.

Segurança Para garantir a segurança do sistema é necessário garantir a seguinte propriedade: transacções que terminam com sucesso na base de dados replicada, fazem evoluir as réplicas para os mesmos estados e observam uma semântica *Snapshot Isolation*.

A garantia desta propriedade assenta nas garantias dadas pelo algoritmo de replicação PBFT. Em particular, o serviço de replicação fornecido por esta biblioteca é equivalente a um único servidor correcto que executa cada operação sequencialmente.

A correcção deste algoritmo depende de dois invariantes: (1) as operações submetidas através da biblioteca de replicação BFT (*begin_trx* e *commit/rollback*) são executadas segundo uma ordem total em todas as réplicas; (2) a resposta a essas operações que é recebida pelo cliente do sistema é a mesma que seria produzida por uma réplica correcta.

Estas duas propriedades são uma consequência do facto de tais operações serem executadas através da biblioteca BFT com a propriedade de atomicidade referida anteriormente. A propriedade (2) é também uma consequência do facto do resultado destas operações apenas depender da sequência de operações de *begin_trx* e *commit* que acontecerem anteriormente, como discutido anteriormente.

Dados estes dois invariantes, a prova de que o sistema obedece a uma semântica SI deduz-se do facto de cada réplica correcta aplicar as operações *begin_trx* e *commit* na sua base de dados local, que garante semântica SI, e devolve o resultado obtido, e ainda porque a operação de *commit* permite validar todos os resultados que o cliente recebeu e aplicar as alterações efectuadas pelo cliente durante a transacção. Como nas várias réplicas a operação de *commit* é executada pela mesma ordem, garante-se que as réplicas correctas evoluem para o mesmo estado.

Focando agora as operações só de leitura, estas também se encontram de acordo com a semântica SI, uma vez que estas transacções iniciam a sua execução com recurso a uma operação PBFT (que grava o *snapshot* corrente do estado da base de dados e que irá ser utilizado para todas as operações da transacção) que se encontra na mesma posição da sequência de operações executadas em todas as réplicas correctas, como referido anteriormente. Como os valores são verificados por f+1 réplicas, existe pelo menos uma réplica correcta nesse conjunto que irá retornar o resultado correcto de acordo com a semântica SI.

Progresso Para garantir o progresso do sistema é necessário garantir a seguinte propriedade: operações que são iniciadas por um cliente são eventualmente executadas pelo sistema.

Para garantir esta propriedade, são necessárias as mesmas assunções que foram necessárias para garantir a mesma propriedade do protocolo PBFT, ou seja, que os atrasos na recepção das mensagens não cresça superlinearmente.

Assumindo a propriedade progresso do protocolo PBFT, podemos garantir que as operações de *begin_trx*, *commit* e *rollback* são eventualmente executadas. Adicionalmente, as operações que não são executadas através da biblioteca PBFT são simples RPCs que são eventualmente executados sob o mesmo conjunto de requisitos. A execução de nenhuma destas operações tem condições para bloquear devido à construção do algoritmo e assim se garante que todas as operações dos clientes serão executadas eventualmente.

3.7 Optimizações

Para permitir uma melhor avaliação das contribuições desta dissertação, nesta secção discutem-se as alterações propostas neste trabalho aos algoritmos do sistema Byzantium quando comparados com os algoritmos propostos na primeira versão do sistema [20]. Estas modificações ao algoritmo constituem as principais contribuições desta dissertação ao nível do desenho da solução. A nova implementação do protótipo apresentada no próximo capítulo, constitui igualmente uma contribuição deste trabalho.

3.7.1 Propagação das operações

No algoritmo original, quando uma operação era invocada no cliente, ela apenas era enviada à réplica responsável pela transacção. Como resultado dessa decisão, era necessário enviar a lista das operações a todas as outras réplicas no momento do *commit*. Na versão actual, como explicado, todas as operações são propagadas imediatamente para todas as réplicas. Esta propagação das operações durante a sua execução permite reduzir a dimensão da mensagem de *commit*, que contém apenas um *hash* das operações e resultados para verificação de consistência. Reduzir o tamanho da mensagem de *commit* é tanto mais importante quanto maior for a dimensão da transacção. Uma grande dimensão podia causar problemas à utilização de mensagens *multicast*, obrigando à execução de um protocolo de transferência que particionasse as mesmas. Adicionalmente, como a propagação é efectuada usando mensagens *multicast* não impõe sobrecarga adicional nos clientes. Nos servidores que não o responsável pela transacção, o processamento destas mensagens também não impõe sobrecarga significativa.

Esta alteração permite a implementação da optimização descrita de seguida. No entanto, esta alteração teve implicações na forma de validar as transacções na presença de um cliente bizantino, levando à adopção da solução descrita na secção 3.6.2.

3.7.2 Redução da latência nas transacções de leitura

Devido à optimização anterior, todas as operações chegam a todas as réplicas aproximadamente no momento em que estão a ser executadas. Assim, é extremamente conveniente aproveitar essa situação para executar algumas dessas operações. No geral, todas as operações de leitura de qualquer transacção, que se encontrem antes da primeira operação de escrita, podem ser imediatamente executadas e o respectivo resultado devolvido ao cliente. Note-se que nesta definição recaem as transacções só com leituras, em particular.

No lado do cliente, após uma operação de leitura ser submetida às réplicas e o primeiro resultado ter chegado, a aplicação pode continuar a execução com o resultado especulativo. Todas os outros resultados são recolhidos em *background*. Quando o cliente pretende terminar a operação, a análise dos resultados recebidos permite validar localmente a correcção da execução, caso o resultado de cada operação seja confirmado por pelo menos f+1 réplicas diferentes. Neste caso, a operação de *commit* pode retornar imediatamente, sendo apenas necessário terminar a transacção nas outras réplicas (algo que pode ser efectuado em *background* também).

Esta optimização não era possível na versão anterior do algoritmo porque as operações

apenas eram propagadas para as réplicas secundárias no momento do *commit*. Apenas nessa altura seriam executadas o que impunha um atraso significativo no *commit* da transacção, ao contrário da solução apresentada.

Por outro lado, o facto de apenas serem necessários f+1 resultados de réplicas diferentes para cada operação leva-nos à optimização seguinte.

3.7.3 Redução do número de réplicas que executam operações de leitura

Nesta versão do algoritmo não é necessário declarar uma transacção de leitura como tal. Todas as transacções quando são iniciadas são assumidas como transacções de leitura, e no momento da primeira ocorrência de uma escrita, a transacção é promovida a transacção de escrita e tratada dessa forma até ter sido terminada. Tendo esse facto presente, é necessário ter em consideração um resultado importante. Como as operações de leitura não alteram o estado da base de dados não é necessário que sejam executadas em todas as réplicas. Na realidade, apenas f+1 réplicas do sistema são necessárias para executar uma operação de leitura, caso não ocorram falhas.

Assim, é possível reduzir o número de réplicas que executam as operações de leitura descritas na optimização anterior, reduzindo a carga em cada uma das réplicas.

Na versão original do algoritmo todas as operações de qualquer transacção eram executadas em todas as réplicas do sistema, incluindo transacções só com leituras. As melhorias efectuadas permitem que o sistema implementado apresente bons resultados na execução de transacções de leitura, como se apresenta na avaliação do protótipo.

```
1   function db_begin() : txHandle
2       uid = generate new uid
3       (coordRep, secondReps) = select( replica, set of f replicas )
4       BFT_exec( <begin_trx, uid, (coordRep, secondReps)> )
5       ops = new Map
6       readOnly = true
7       opCount = 0
8       trxHandle = new trxHandle( uid, coordRep, secondReps, opCount, ops, readOnly )
9       return txHandle
10  end function
11
12  function db_read_op( txHandle, op ) : result
13      opNum = ++txHandle.opCount
14      mcast( <read( txHandle.uid, opNum, op )> )
15      recv( <readResult( txHandle.uid, opNum, HOp, res )> ) // 1st result
16      txHandle.ops.put( opNum, <op, 1, H(res), 'read'> )
17      return res
18  background // additional results
19      recv( <readResult( txHandle.uid, opNum, HOp, res )> )
20      <count,HRes> = txHandle.ops.get( opNum )
21      if( HRes == H( res ) )
22          trxHandle.ops.put( opNum, <op, count+1, HRes, 'read'> )
23      endif
24  end function
25
26  function db_write_op( txHandle, op ) : result
27      opNum = ++txHandle.opCount
28      mcast( <write( txHandle.uid, opNum, op )> )
29      recv( <writeResult( txHandle.uid, opNum, HOp, res )> )
30      txHandle.readOnly = false
31      txHandle.ops.put( opNum, <op, 1, H(res), 'write'> ) ;
32      return res
33  end function
34
35  function db_commit( txHandle )
36  concurrent
37      if( readOnly )
38          while LastReadConfirmed( trxHandle ) < txHandle.opCount
39              wait
40          end while
41          return
42      endif
43  with
44      lastConfirmed = LastReadConfirmed( txHandle )
45      HOps = H( ListOps( txHandle ) )
46      HRes = H( ListRes( txHandle, lastConfirmed ) )
47      res = BFT_exec( <commit, txHandle.uid, lastConfirmed( txHandle ), HOps, HRes> )
48      if( res == true )
49          return
50      else
51          throw ByzantineExecutionException
52      endif
53  end concurrent
54  end function
```

Figura 3.2 Algoritmo executado pelos clientes do sistema Byzantium.

```
1     upcall for BFT_exec( <begin_trx, uid, (coordReplica, secondReplicas)> )
2         DBTxHandle = db.begin()
3         // variables represent per-trx state
4         ops = new Map
5         readOnly = true
6         txSrvHandle = new txSrvHandle( uid, DBTxHandle, coordRep, secondReps, ops, readOnly )
7         openTrxs.put( uid, trxSrvHandle )
8     end upcall
9
10    upcall for BFT_exec( <commit, uid, lastConfirmed, cltHOps, cltHRes> ) : boolean
11        txSrvHandle = openTrxs.get( uid )
12        openTrxs.remove( uid )
13        if( txSrvHandle.coordRep != THIS_REPLICA )
14            execOK = exec_and_verify( txSrvHandle.DBTxHandle, lastConfirmed,
15                txSrvHandle.ops, cltHOps, cltHRes )
16            if( NOT execOK )
17                DBTxHandle.rollback()
18                return false
19            endif
20            return DB_trx_handle.commit()
21        endif
22    end upcall
23
24    upcall for recv( <read( uid, opNum, op )> )
25        txSrvHandle = openTrxs.get( uid )
26        txSrvHandle.ops.put( opNum, <op,-> )
27        if( txSrvHandle.coordRep == THIS_REPLICA OR
28        ( txSrvHandle.readOnly AND txSrvHandle.secondReps.contains( THIS_REPLICA ) ) )
29            result = txSrvHandle.DBTxHandle.exec( op )
30            txSrvHandle.ops.put( opNum, <op,result> )
31            send_reply( <readResult( txSrvHandle.uid, opNum, H( op ), result )> )
32        endif
33    end upcall
34
35    upcall for recv( <write( uid, opNum, op )> )
36        txSrvHandle = openTrxs.get( uid )
37        txSrvHandle.ops.put( opNum, op )
38        if( txSrvHandle.coordReplica == THIS_REPLICA )
39            result = txSrvHandle.DBTxHandle.exec( op )
40            txSrvHandle.ops.put( opNum, <op,result> )
41            send_reply( <writeResult( uid, opNum, H( op ), result )> )
42        endif
43    end upcall
```

Figura 3.3 Algoritmo executado pelas réplicas do sistema Byzantium.

4 . Implementação da solução apresentada

Este capítulo contém a descrição da implementação do sistema, descrevendo o modo como foram implementados os diferentes componentes referidos no capítulo anterior. Além disso são também discutidas algumas possíveis optimizações. Inicialmente são apresentados conceitos gerais acerca da implementação do sistema, seguindo-se a descrição das implementações dos componentes de comunicação fiável e da biblioteca de replicação bizantina. Por último, descreve-se a implementação do *driver* JDBC criado, na qual são implementados os algoritmos propostos. O objectivo deste capítulo não consiste em apresentar exaustivamente a implementação efectuada, mas antes descrever partes não triviais necessárias à constituição da solução apresentada.

4.1 Apresentação geral

A implementação do sistema foi realizada na totalidade em Java 1.6. Os componentes do sistema Byzantium, assim como a biblioteca de replicação bizantina e o módulo de comunicação FIFO são implementados em Java. As réplicas utilizam múltiplas *threads* como forma de aumentar o desempenho do sistema ao permitir a execução de operações em concorrência. Assim, cada réplica tem múltiplas *threads* a tratar das mensagens enviadas directamente dos clientes usando o mecanismo de comunicação FIFO e uma *thread* trata as operações submetidas pela biblioteca de replicação bizantina.

Esta versão do sistema utiliza o sistema de gestão de base de dados PostgreSQL 8.3.4. Para aceder ao sistema de gestão de bases de dados cada réplica utiliza o *driver* JDBC incluído nessa versão. Em cada uma das réplicas é mantida uma cópia total da base de dados. Tal como discutido no capítulo 3 o algoritmo implementado garante a convergência eventual das réplicas.

4.2 Módulo de comunicação FIFO

Em redes de computadores, a comunicação é um ponto essencial. Como tal, é importante que existam componentes de comunicação que garantam algumas propriedades, nomeadamente fiabilidade. De modo a introduzir no sistema a capacidade de comunicar em grupo, especialmente

quando se pretende distribuir uma mensagem por todas as réplicas, foi implementado um componente de comunicação que permite envio de mensagens ponto a ponto ou destinadas a um grupo previamente constituído segundo uma semântica FIFO. Este componente tem como base o *multicast* convencional, ao qual foi adicionado um mecanismo de recuperação de falhas. Uma vez que as réplicas se encontram a executar num ambiente em que dispõem de uma ligação à rede de boa qualidade (rede local), é esperado que a perda de mensagens seja praticamente inexistente. No entanto, caso alguma mensagem não seja entregue em um dos destinos, o módulo tem um mecanismo para detectar e corrigir a situação.

Tal como no *multicast*, este módulo de comunicação implementa o conceito de grupo. As mensagens enviadas para o grupo são entregues em todos os membros do grupo, pelo que todos os participantes interessados em receber as mensagens devem juntar-se previamente. No caso do sistema Byzantium, todas as réplicas juntam-se ao mesmo grupo, enquanto que os clientes mantêm-se fora do grupo. Cada participante mantém um contador que lhe permite efectuar a numeração das mensagens enviadas. Além disso, mantém também uma estrutura de dados que contém o último número de série recebido de cada um dos outros participantes, incluindo elementos do grupo e clientes, e um registo temporário das mensagens enviadas recentemente. Esta informação permitirá detectar uma mensagem em falta e corrigir esse problema. No momento do envio, a mensagem a enviar é encapsulada num pacote próprio que permite o envio de informação de controlo do *socket*. Essa mensagem é guardada no emissor com o propósito de a retransmitir caso ocorra algum problema pontual na comunicação até algum dos outros participantes.

Quando uma mensagem é recebida, o receptor limita-se a consultar o número de série da mesma (*serial*), assim como o último número de série recebido do mesmo emissor (*last_serial*), decidindo a próxima acção a desempenhar. Caso $serial < last\_serial+1$, então a mensagem é antiga, e já foi entregue à aplicação. Neste caso a mensagem é descartada. Esta situação pode ocorrer caso duas mensagens cheguem ao destino por ordem inversa, e um pedido de retransmissão seja encaminhado (originando uma retransmissão da mensagem desnecessário). Se $serial = last\_serial+1$, então a mensagem recebida é a próxima a ser entregue, e é colocada na fila de mensagens a serem consumidas. Por último, se $serial > last\_serial+1$, estamos na situação em que pelo menos uma mensagem enviada pelo emissor da mensagem não chegou a este participante, e como tal, é enviada uma mensagem a requisitar a retransmissão da mesma. O emissor, ao receber esta mensagem, volta a retransmitir a mensagem perdida apenas para quem a requisitou. Quanto às mensagens que chegam nestas condições, são guardadas enquanto

```
1           if( message.getSerialNumber() == acks[message.getSenderId()] + 1 )
2           {
3               // is next to deliver
4               try { toDeliver.put( message.getDatagram() ) ; }
5               catch( InterruptedException e ) { e.printStackTrace() ; }
6               acks[message.getSenderId()]++ ;
7               // check if any of the pending messages is the next to deliver
8           }
9           else if( message.getSerialNumber() < acks[message.getSenderId()] + 1 )
10          {
11              // discard. message was already delivered
12          }
13          else // message.getSerialNumber() > acks[message.getSenderId()] + 1
14          {
15              // send nacks for missing packets
16              for( int i = acks[message.getSenderId()] + 1 ; i < message.getSerialNumber(); i++ )
17              {
18                  // create and send nack packets for all missing packets (i).
19              }
20          }
```

Figura 4.1 Algoritmo de decisão utilizado na recepção de mensagens fiáveis.

aguardam que as mensagens anteriores sejam entregues de modo a cumprir a ordem de emissão.

Assim como a implementação da primitiva de comunicação em grupo, existe uma implementação de uma operação de comunicação ponto-a-ponto que permite detectar a perda de mensagens através do envio de *acknowledgements* O emissor, caso não receba um *acknowledgement* da mensagem enviada, volta a enviá-la ao fim de um dado período de tempo. Como este componente será utilizado numa rede local de qualidade e com pouco tráfego concorrente, é esperado que na realidade nunca se percam mensagens.

Este conjunto de alterações conferem ao sistema de comunicação as seguintes propriedades: fiabilidade ponto-a-ponto e entrega das mensagens de cada tipo por ordem FIFO.

A implementação deste componente recorre a uma *thread* dedicado ao atendimento do *socket*. As mensagens que chegam são analisadas e processadas conforme o tipo que possuem: mensagens do grupo, mensagens ponto-a-ponto e *nacks*. Este passo permite tratar mensagens em falta ou ordenar as mesmas pelo ordem pela qual foram enviadas. As mensagens de cada tipo que se encontram em condições de serem entregues à aplicação são colocadas numa estrutura de dados FIFO, e posteriormente consumidas pelos outras *threads* no sistema. Esta estrutura coordena acessos concorrentes de modo a lidar com a situação de múltiplos *threads* a consumir mensagens recebidas.

4.2.1 Autenticação e controlo de integridade

A biblioteca desenvolvida permite ainda efectuar opcionalmente a autenticação do emissor da mensagem e controlar a integridade da mesma. Esta funcionalidade é implementada recorrendo a HMAC [3], assumindo que cada par de parceiros de comunicação partilham uma chave simétrica. Assim, a cada mensagem são adicionadas as assinaturas digitais dos destinatários da mensagem. Na recepção, a assinatura é verificada e a mensagem é descartada caso tenha sido alterada durante a transmissão. Esta funcionalidade é usada pela biblioteca de BFT para verificar a autenticidade das mensagens de forma transparente.

4.3 Java BFT

Tal como referido anteriormente, o sistema implementado tira partido de um componente que fornece a possibilidade de executar operações tolerantes a falhas bizantinas. Nesta versão do sistema Byzantium, essa biblioteca PBFT encontra-se implementada em Java. A biblioteca é composta por dois componentes, um que se encontra no servidor, e outro no cliente.

No lado do cliente, é oferecida uma interface que permite a execução das operações. Para executar uma operação, a aplicação deve compor um objecto que implemente a interface *RSOperation*.

A implementação desta biblioteca tem como base uma versão disponibilizada por uma equipa do MIT, que implementa parcialmente o algoritmo descrito em [5]. A biblioteca fornecida era baseada em comunicação ponto-a-ponto TCP. Uma avaliação inicial de desempenho mostrou que esta aproximação tinha um desempenho insatisfatório, pela necessidade de enviar 3f+1 mensagens por cada difusão de mensagens para o grupo. Assim, esta biblioteca foi modificada para integrar a utilização do módulo de comunicação fiável descrito anteriormente. Cada uma das fases do algoritmo (*preprepare*, *prepare* e *commit*) recorre à primitiva de comunicação em grupo do módulo de comunicação fiável.

A biblioteca usada apresenta algumas limitações face à versão completa do PBFT [5]. A primeira limitação é a ausência do mecanismo de checkpoints. Este mecanismo é usado no algoritmo original para permitir que uma réplica que se atrase na execução das operações obtenha o estado das réplicas mais avançadas sem executar todas as operações. A utilização do mecanismo de comunicação FIFO elimina parcialmente esta necessidade, pois garante-se que durante a execução normal, as réplicas não se atrasam. Caso a réplica primária seja bizantina

é possível que uma réplica correcta se atrase na execução das operações. Por exemplo, pode dar-se o caso em que a réplica primária não envie as operações para essa réplica correcta. Para lidar com este problema seria necessário usar o mecanismo de *checkpoints* e transferência de estado do protocolo PBFT, que no caso do Byzantium poderia consistir num sufixo da sequência de transacções executadas. A segunda limitação prende-se com a não implementação do mecanismo de decisões não deterministas, o que impede a implementação completa do algoritmo descrito na secção 3.6.2 para lidar com clientes bizantinos.

De notar que estas limitações apenas são significativas na presença de nós maliciosos, dado que a combinação de mensagens necessárias para provocar a ocorrência do problema é improvável num nó que esteja a comportar-se arbitrariamente sem intenção de causar danos.

4.4 Cliente Byzantium

A implementação do módulo cliente do Byzantium, consiste na implementação de um *driver* JDBC. Para criar um *driver* com a capacidade de executar transacções, é necessário implementar pelo menos quatro interfaces: *java.sql.Driver*, *java.sql.Connection*, *java.sql.Statement* e *java.sql.ResultSet*. Nas implementações feitas no âmbito deste projecto, as quatro classes que implementam estas interfaces têm acesso à biblioteca de replicação bizantina e ao módulo de comunicação em grupo, conferindo-lhes a capacidade de serializar operações segundo o algoritmo PBFT. A implementação do *driver* consiste em adaptar as operações relativas ao modelo transaccional, isto é, *begin_trx*, *commit*, *rollback* e operações de leitura e escrita, aos métodos existentes nas interfaces JDBC.

4.4.1 Driver

A interface *java.sql.Driver* é aquela que todos os *drivers* JDBC devem implementar. Esta é composta por seis métodos relevantes, dos quais quatro são apenas métodos de consulta de informação acerca do driver. Os métodos *getMajorVersion* e *getMinorVersion* retornam a versão do driver. O método *jdbcCompliant* permite saber se o *driver* implementado é genuinamente compatível com JDBC, enquanto que o método *getPropertyInfo* permite consultar informação acerca das propriedades do *driver*. Além destes, existem mais dois métodos, um pouco mais interessantes para a execução de transacções: *acceptsURL* e *connect*. O primeiro permite testar

se um dado URL é aceite por este *driver*. A implementação deste método basicamente consiste em verificar se o URL indica o sub-protocolo Byzantium. O segundo método é aquele que permite criar conexões para a base de dados. O resultado da invocação deste método será um objecto da classe que implementa a interface *java.sql.Connection*. O URL deve corresponder ao formato *jdbc*:*byzantium2*:*dbname*, onde os dois primeiros atributos indicam o protocolo principal (JDBC) e o sub-protocolo (segunda versão deste sistema). A terceira parte do URL permite indicar o nome da base de dados a utilizar.

No caso do sistema Byzantium, a operação *connect* é enviada para todas as réplicas através da biblioteca de replicação bizantina de modo a garantir que todas as réplicas se encontram em condições de abrir uma conexão em cada um dos sistemas de gestão de base de dados. Terminada a operação, um objecto *Connection* é devolvido à aplicação, sabendo-se que as réplicas se encontram disponíveis para iniciar a execução de transacções. Na realidade, esta operação de *connect* é o mapeamento directo da operação de início de transacção *begin_trx*.

4.4.2 Connection

A interface *java.sql.Connection* é o conceito mais próximo de transacção que existe no JDBC. Após criada, os métodos disponibilizados permitem criar e preparar operações (*createStatement*, *prepareStament* e *prepareCall*). São os objectos retornados por estes métodos que irão executar de facto as operações. A classe que implementa a interface *java.sql.Connection* é também a responsável por executar as operações de *commit* e de *rollback*. Tal como na definição do modelo transaccional, estas operações têm o propósito de indicar como terminará a transacção. Em caso de *commit*, todas as operações realizadas deverão tornar-se definitivas enquanto que no caso de *rollback*, essas mesmas operações serão descartadas, ficando o estado final igual ao inicial. As duas operações são executadas em todas as réplicas, concluindo a transacção corrente e iniciando uma nova transacção. Assim, a operação JDBC de *commit* (e de *rollback*), executa os passos para terminar e iniciar a transacção seguinte (*commit* + *begin_trx* da transacção).

No entanto, é necessário indicar um pormenor acerca do modo como o JDBC executa uma transacção. Ao contrário daquilo que é efectuado no Byzantium, no JDBC uma transacção apenas é iniciada no momento em que a primeira operação é efectuada. Por isso, apesar de a operação de início de transacção ser executada após ser serializada pela biblioteca de replicação bizantina, não existe garantia de que a transacção de facto se inicie exactamente no mesmo

estado em todas as réplicas. Para contornar esta situação, é necessário forçar o início da transacção na base de dados aquando da execução da operação de início de transacção. Para isso basta efectuar uma leitura de um qualquer valor de uma tabela privada do sistema, criada para o efeito. Esta solução é semelhante à utilizada em sistemas apresentados anteriormente como é o caso do GSI [9] ou do Tashkent [7].

A junção das duas operações (execução do *begin_trx* logo após cada *commit*) faz com que a transacção seguinte seja iniciada mais cedo do que o necessário, aumentado a probabilidade de conflito entre transacções. Por outro lado, esta decisão permite reduzir o número de operações efectuadas através da biblioteca BFT (que utiliza um número superior de mensagens por operação). O modo como as transacções são tratadas no sistema Byzantium poderia ser reconfigurado, explorando as diferentes possibilidades, nomeadamente, separando as operações de *commit* e *begin_trx*, e efectuando a operação *begin_trx* da transacção seguinte juntamente com a primeira operação da mesma.

4.4.3 Statement

A interface *java.sql.Statement* representa uma operação. Os métodos mais significativos são aqueles que permitem executar operações sobre a base de dados: *executeQuery* e *executeUpdate*. O primeiro método permite a execução de operações do tipo *select*, isto é, operações de leitura; enquanto que o segundo método tem a capacidade de executar operações do tipo *update*, *insert* e *delete*. As operações SQL são passadas como argumento destes métodos, e propagadas para as réplicas afim de serem executadas. A execução das operações segue os algoritmos descritos anteriormente. O processamento de todas as respostas recebidas dos servidores é efectuado por uma *thread* autónoma. Assim, a *thread* the envia uma operação de leitura ou de escrita bloqueia-se até que a *thread* autónoma indique a recepção do resultado esperado. Esta aproximação é necessária para lidar com as múltiplas respostas de uma operação de leitura.

Depois de o primeiro resultado ser recebido, é criado um ResultSet que contém o resultado da operação, tipicamente um conjunto de linhas (ou tuplos). No entanto, como o resultado de uma operação *select* pode conter um número de linhas grande, faz sentido transferir essa informação por partes e conforme vai sendo acedida. Para isso, juntamente com a mensagem de resposta à operação contém apenas algumas linhas e informação acerca da existência de seguintes. Com estes dados, é possível evitar a comunicação nos acessos às primeiras linhas do resultado e determinar se existem linhas adicionais no ResultSet sem que seja necessário enviar

uma mensagem de propósito para o efeito, isto é, uma execução do método *next* com resultado negativo. Para atingir este objectivo, o ResultSet no lado da réplica é consultado imediatamente após a sua criação e essa informação é devolvida ao cliente.

Por outro lado, duas outras optimizações na gestão do conjunto dos resultados também foram introduzidas. A descrição em maior detalhe encontra-se na subsecção 4.4.4.

4.4.4 ResultSet

Esta interface representa uma sequência de resultados obtidos aquando da execução de uma operação de leitura. Tipicamente, uma operação SQL do tipo *select* resulta numa colecção de tuplos. A interface *java.sql.ResultSet* permite a interacção com essa colecção de tuplos, disponibilizando métodos para iterar os diferentes items e consultar as colunas devolvidas. A nova classe que implementa esta interface não se limita a propagar as operações para as réplicas. Em vez disso, foram implementadas duas optimização de modo a reduzir a necessidade de comunicação. Primeiro, a classe mantém a linha corrente no lado do cliente, permitindo a consulta de todos os valores sem que seja necessário enviar qualquer mensagem (por exemplo através de *getInt* ou *getString*). A complexidade da implementação encontra-se em torno do método *next*. Sempre que o método *next* é invocado com o propósito de fazer avançar a iteração, a linha seguinte é completamente transferida da réplica responsável para o cliente. Segundo, tanto a operação *executeQuery* como a operação *next*, sempre que necessitam de comunicar com as réplicas para obter resultados, transferem não apenas uma linha mas um conjunto pequeno de linhas do resultado (cinco linhas por omissão). Assim, expande-se a optimização descrita anteriormente de forma a evitar mais algumas trocas de mensagens.

Por fim, após analisar alguns resultados experimentais, chegou-se à conclusão que a operação *close* de um ResultSet estava a ser invocado um número significativo de vezes, e que o facto de ser necessário trocar mensagens para a executar estava a penalizar o desempenho do sistema. Desde modo, e como apenas existe um ResultSet para cada Statement, optou-se por fechar cada ResultSet na operação *executeQuery* seguinte (isto é, antes da criação do ResultSet seguinte). O objecto é fechado localmente no lado do cliente em definitivo, e permanece aberto no lado do servidor até a próxima operação de leitura ser executada. Como não existe forma de o cliente se aperceber desta ligeira inconsistência temporária, considera-se que a redução no número de mensagens trocadas e no tempo de execução das transacções justifica a decisão.

4.5 Servidor Byzantium

O servidor Byzantium implementa o algoritmo apresentado no capítulo anterior. Cada réplica mantém informação acerca de cada conexão aberta. Associada a cada conexão, existe um conjunto de Statements e respectivos ResultSets. Por decisão de implementação, existe também uma *thread* própria para cada conexão, responsável por tratar as mensagens relativas à mesma. Para cada conexão de um cliente, o servidor mantém a informação do estado actual da transacção e uma conexão JDBC para a base de dados local.

A execução das várias operações corresponde a uma implementação simples dos algoritmos apresentados usando a conexão JDBC para a base de dados local. A única situação que merece destaque prende-se com a resolução de potenciais conflitos relativos à execução concorrente de múltiplas transacções, que se discute de seguida.

4.5.1 Resolução de conflitos (bases de dados com *locks*)

Neste ambiente em que se pretendem executar transacções em várias réplicas em simultâneo, existe a necessidade de garantir que não existem problemas de concorrência. Em particular, é necessário garantir que a execução das várias transacções nas várias réplicas se processa de modo idêntico. Os sistemas de gestão de bases de dados que efectuam o seu controlo de concorrência de modo optimista sem recurso a *locks*, não constituem um problema. Neste caso, como o momento do *commit* e do *rollback* de todas as transacções é executado pela mesma ordem em todas as réplicas, o resultado obtido será idêntico em todos as réplicas. O mesmo não é verdade para os sistema baseados em *locks*, como se discute de seguida.

O problema acontece porque todas as réplicas podem ser eleitas como réplicas responsáveis de transacções. Assim existe uma ligeira diferença entre executar as operações como réplica responsável, e executar as operações como réplica secundária (no momento do *commit*). Chamemos transacções locais de uma dada réplica às transacções cuja réplica responsável é essa réplica, e transacções remotas às restantes.

As operações das transacções locais são executadas normalmente, sendo controladas pelo mecanismo de concorrência do sistema de gestão de base de dados local. O problema surge quando uma transacção remota de escrita pretende executar a operação de *commit*. Num sistema de gestão de base de dados baseado em *locks*, se a transacção remota necessitar de algum recurso que entretanto tenha sido reservado por transacções locais, estamos perante uma situação de

```
1   CREATE TRIGGER trig_tableName
2   AFTER INSERT ON tableName
3       FOR EACH ROW EXECUTE PROCEDURE trigfunc_tableName();
4
5   CREATE OR REPLACE FUNCTION trigfunc_tableName() RETURNS trigger AS '
6   BEGIN
7       RAISE NOTICE ''tableName:%'', NEW.pkName;
8       RETURN NULL;
9   END;
10  ' LANGUAGE 'plpgsql';
```

Figura 4.2 Exemplo de *trigger* de captura de identificadores.

bloqueio. Para garantir a correcção da solução é necessário que o *commit* da transacção remota seja o próximo a ser executado. No entanto, do ponto de vista do sistema de gestão de base de dados, isso só é possível se as transacções locais em conflito terminarem de alguma forma.

A ideia para resolver este problema consiste em abortar as transacções locais que impedem a conclusão da transacção remota que pretende terminar. Para isso, capturam-se os identificadores dos registos acedidos, associando-os às transacções correspondentes. Com esta informação é possível determinar possíveis conflitos entre uma transacção local e uma remota antes de executar a transacção remota.

Para recolher estes identificadores, recorreu-se à instalação de um conjunto de *triggers* na base de dados. O funcionamento desses *triggers* consiste simplesmente em enviar uma mensagem de erro contendo o par (chave primária, tabela) para cada linha consultada. Essas mensagens são capturadas através de JDBC no momento da execução das operações nas réplicas, e o seu conteúdo é armazenado no conjunto de identificadores associados à operação executada.

Assim, no momento de executar o *commit* de uma transacção remota, os conjuntos de identificadores associados à mesma são comparados com os conjuntos de identificadores das transacções locais. Para decidir se é necessário abortar uma transacção local, efectua-se a intersecção entre o conjunto de identificadores respectivo e conjunto de identificadores da transacção remota e caso o resultado seja não vazio, a transacção local é abortada. Como indicado previamente, esta técnica permite garantir que a operação de *commit* da transacção remota é executada sem que ocorram bloqueios.

Esta solução possui um problema. Existe uma situação na qual o comportamento obtido pode não ser o desejado. Como o modelo de falhas considerado é o modelo de falhas bizantinas, é necessário ter em conta o caso em que a réplica responsável pela transacção pode ser uma réplica bizantina, e o processo de recolha, processamento e envio da informação necessária para controlar a ocorrência de bloqueios se encontra comprometido. Assim, é possível que

uma réplica nestas condições, ao enviar informação errada, provoque uma de duas situações indesejáveis. Primeiro, caso o conjunto de pares <*tabela, chave primária*> esteja incompleto, a réplica que pretende efectuar o *commit* remoto pode concluir que não existem bloqueios e iniciar a execução de operações que irão bloquear. Segundo, pode-se dar o caso em que esse conjunto de pares contenha entradas em excesso, fazendo com que transacções sejam abortadas desnecessariamente.

Este problema deve-se ao facto de essa informação ser capturada por uma só réplica, e não existir nenhum tipo de verificação ou consenso. Para o resolver, é necessário verificar a existência de conflitos sem recorrer ao envio dos conjuntos de escritas. Mantendo os *triggers* de captura de identificadores, cada réplica colecciona o conjunto das escritas efectuadas por cada transacção pela qual a réplica é responsável. No entanto, essa informação não é partilhada com nenhum outro participante no sistema. No momento do *commit* de uma transacção remota, de modo a evitar um bloqueio durante a execução das operações de escrita, a réplica utiliza a informação previamente recolhida (referente às transacções locais em curso) em conjunto com o código SQL das operações de escrita, para determinar a existência de conflitos antes de executar as operações. Se a operação de escrita incluir a chave primária da tabela (seja um *insert*, um *update* ou um *delete*) a resolução do problema é relativamente trivial, pois resume-se a efectuar a comparação dos valores. Caso isso não se verifique, é necessário consultar a tabela através de uma operação de *select* para obter os identificadores de cada linha bloqueada. Note-se que todas as operações realizadas sobre o código SQL podem ser realizadas no momento em que a réplica recebe a operação, não implicando qualquer atraso no momento do *commit*. Esta forma de evitar o bloqueio é semelhante à descrita anteriormente, a diferença reside no facto de não existir partilha de informação entre as réplicas que pode induzir um comportamento indesejável.

O modo como se verifica a existência de *deadlocks* pode ainda ser melhorado. Com base na informação recolhida, isto é, os conjuntos de identificadores dos registos acedidos em cada transacções podem ser utilizados para compor uma *query* do tipo *select for update*. Uma *query* deste tipo devolve exactamente o mesmo resultado que um *select* normal, no entanto, tem a particularidade de bloquear todos os registos acedidos. Com uma *query* com estas características é possível efectuar a verificação da disponibilidade dos recursos necessários e garantir que os mesmo estarão disponíveis para a execução das operações da transacção que pretende terminar.

5. Avaliação

Este capítulo contém uma descrição dos testes realizados ao protótipo implementado, assim como os resultados obtidos e respectiva análise. O objectivo dos testes consiste em avaliar o desempenho dos algoritmos propostos num cenário realista.

5.1 Ambiente de avaliação

5.1.1 Benchmark

De modo a realizar os testes, foi usada uma implementação *open-source* do *benchmark standard* de processamento de transacções *online* TPC-C[1]. Apesar de esta implementação não ser exactamente igual ao *standard* do TPC-C, permite ainda assim fazer uma avaliação do sistema num cenário realista, e avaliar o *overhead* da solução apresentada. O *benchmark* utilizado apresenta ligeiras modificações em relação ao original, nomeadamente, a existência de uma fase de aquecimento (ou *warm-up*) e a execução de cada cliente num processo autónomo. Esta última característica é necessária devido à biblioteca usada na comunicação entre os componentes no sistema. A fase de aquecimento introduzida consiste em executar transacções normalmente, com o objectivo de preencher as *caches* do sistema de base de dados colocando os componentes do sistema num estado aproximado daquele que se verifica durante a execução do *benchmark*. Assim, este mecanismo permite evitar que sejam recolhidos os resultados relativos à fase de inicial do *benchmark*, momento em que todas as *caches* utilizadas se encontram vazias.

Como já foi referido, o TPC-C é um *benchmark* de processamento de transacções *online*. Este *benchmark* simula um ambiente de tratamento de encomendas onde um conjunto de clientes executa transacções sobre uma base de dados previamente criada. Note-se que o TPC-C não se compromete com nenhuma área em particular de qualquer actividade económica. Por outras palavras, representa qualquer indústria que requer a necessidade de gerir as vendas, os *stocks* e a distribuição dos seus produtos ou serviços.

Descrevendo em maior detalhe os conceitos considerados no TPC-C, o modelo empresarial que implementa pode ser visto como uma companhia de distribuição que opera vários armazéns associados a várias zonas de vendas. Cada armazém fornece dez zonas de vendas, e cada zona

[1] `http://sourceforge.net/projects/benchmarksql/`

de vendas serve três mil consumidores. Um operador de uma zona de vendas pode iniciar uma transacção de um dos cinco tipos a qualquer momento. O tipo e a frequência das transacções são modelados a partir de situações reais. Existem cinco tipos de transacções que se apresentam se seguida. Dois desses tipos são muito mais frequentes que os restantes: a transacção de registo de uma nova encomenda, tipicamente com cerca de dez items, e a transacção de registo do pagamento de uma encomenda. Os restantes tipos de transacção são a consulta do estado de uma encomenda previamente efectuada, o processamento de entrega de um conjunto de dez encomendas e a consulta do *stock* de um armazém.

A carga utilizada no TPC-C consiste maioritariamente em transacções de leitura e escrita, e onde apenas uma pequena percentagem são transacções só com leituras. Nas secções seguintes deste capítulo encontram-se os resultados das experiências efectuadas. Primeiro apresentam-se e discutem-se os resultados dos testes efectuados com a carga standard do TPC-C. A seguir, apresenta-se uma secção com os resultados obtidos usando uma carga *read-only*.

5.1.2 Configuração das experiências

As experiências efectuadas tiveram como objectivo principal avaliar o *overhead* relacionado com a incorporação de tolerância a falhas bizantinas num sistema de replicação de bases de dados, assim como analisar a eficácia do algoritmo de replicação proposto. Assim, para alcançar este objectivo, os resultados do protótipo implementado (nomeado como Byzantium) são comparados com três outras soluções desenvolvidas.

Primeiro, uma solução em que o acesso dos clientes é efectuado através de um *proxy* (nomeada como *Proxy*) que acede à base de dados localizada na mesma máquina. Esta solução serve de base de comparação incluindo o *overhead* da criação de uma solução em *middleware*. Este facto é importante porque uma solução *middleware* impõe um nível adicional de indirecção nos pedidos e respostas. Por exemplo, numa operação de leitura, o *middleware* tem de ler o resultado da operação antes de o enviar para o cliente, impondo um atraso na propagação do resultado. Assim, uma comparação directa com o acesso efectuado usando o *driver* da base de dados directamente não estaria a avaliar apenas o *overhead* dos algoritmos mas também o *overhead* da implementação em *middleware* (e da eficiência da mesma).

Segundo, uma solução *proxy* optimizada, onde são incluídas todas as optimizações adicionais realizadas ao algoritmo do sistema Byzantium (nomeada como *OptProxy*). Essas optimizações, descritas em 4.4.3 e 4.4.4 são a não propagação da operação *close* de um ResultSet e

a obtenção de algumas linhas do resultado aquando da execução das operações *executeQuery* e *next*. Note-se que esta versão possui optimizações que não seriam normais num *proxy* convencional onde as operações são simplesmente propagadas para o servidor. O objectivo de incluir esta solução nos testes realizados, tal como a solução apresentada de seguida, é enquadrar o protótipo desenvolvido recorrendo a duas soluções limite. A expectativa seria que o protótipo obtivesse resultados no intervalo entre estas duas soluções.

Terceiro, uma solução em que os clientes executam todas as operações propagando-as para as réplicas usando a biblioteca BFT (nomeada *Full BFT*). Esta solução seria a aproximação imediata para introduzir tolerância a falhas bizantinas num sistema de base de dados. Por outras palavras, a conjugação de qualquer biblioteca de replicação bizantina e um sistema de gestão de bases de dados implica a execução de todas as operações sobre a base de dados através da biblioteca de replicação.

Resumindo, as soluções *Proxy*, *OptProxy*, Byzantium e *Full BFT* diferem principalmente na forma como as operações são propagadas para as réplicas. Necessariamente existem também as diferenças ao nível dos algoritmos implementados pelo Byzantium. Em *Proxy* e *OptProxy* todas as operações que necessitam de comunicação são enviadas apenas a uma réplica; no Byzantium as operações de início e fim das transacções são executadas em todas as réplicas através da biblioteca de replicação bizantina, enquanto que as operações de leitura e escrita são directamente executadas em uma réplica. Por fim, no *Full BFT* todas as operações são propagadas e executadas através da biblioteca de replicação bizantina.

5.1.3 Software de medição

Para obter os tempos de execução das várias operações, permitindo uma avaliação mais profunda dos resultados obtidos usou-se um componente adicional. Este componente é um *driver* JDBC que permite capturar a estatística dos tempos de execução de cada uma das operações efectuadas. Durante o desenvolvimento, a informação obtida através deste *driver* teve particular interesse na medida em que permite identificar operações com alguma margem para optimização. Essas conclusões foram especialmente importantes durante o desenvolvimento do protótipo. A implementação deste *driver* é trivial, tal como é exemplificado na imagem 5.1. Cada operação utilizada é na realidade mascarada por uma operação homónima que se limita a contabilizar o tempo de duração da mesma. Estes dados são mantidos durante a execução e apresentados quando a conexão é fechada.

```
1         public ResultSet executeQuery( String sql ) throws SQLException
2         {
3             long startT = System.nanoTime() ;
4             try
5                 return new BenchResultSet( stmt.executeQuery( sql ), driver ) ;
6             finally
7             {
8                 long endT = System.nanoTime() ;
9                 driver.addValue( BenchDriver.OP_EXEC_QUERY, endT - startT ) ;
10            }
11        }
```

Figura 5.1 Exemplo da implementação de um método do *driver* de *benchmark*.

```
1         sistema operativo:
2             Linux - Kernel v.2.6.18
3         base de dados:
4             PostgreSQL 8.3.4
5         driver JDBC:
6             nativo POstgreSQL (8.3 jdbc4)
7         processadores (quatro máquinas):
8             2 máquinas: 2 x Opteron 275 @ 2.2Ghz (4 cores/nó)
9             2 máquinas: 2 x Opteron 246 @ 2Ghz (2 cores/nó)
10        memória (em cada nó):
11            4 GB
12        interface rede (em cada nó):
13            2 x Gb Ethernet
14            1 x Infiniband 4x (10 Gbps)
15        disco rígido:
16            1 x SATA 80GB
17        rede:
18            Gb Ethernet: SMC8624T (24 ports)
19            Infiniband: Topspin 120 (24 ports IB 4x)
```

Figura 5.2 Especificação do ambiente de execução.

5.1.4 Hardware de testes

Todas as situações de teste foram realizadas com f=1, ou seja, sempre com quatro réplicas no sistema. Cada uma das réplicas do sistema encontrava-se a executar numa das máquinas de um *cluster* com quatro máquinas (ou nós), em que cada máquina contém dois processadores. Duas dessas máquinas estão equipadas, cada uma, com dois processadores *dual-core* AMD Opteron 275 @ 2.2GHz (ou seja, quatro cores). As outras duas têm, cada uma delas, dois processadores *single-core* AMD Opteron 246 @ 2.0GHz (ou seja, dois cores). Cada um dos nós no *cluster* tem 4 GB de memória local, um disco SATA 80GB e uma ligação *ethernet* à rede local a uma velocidade de 1Gbps. Todas as máquinas encontravam-se a executar o sistema operativo Linux, com a versão 2.6.18 do *kernel*. A versão do PostgreSQL utilizada foi a 8.3.4 e a versão da máquina virtual java utilizada foi Sun VM v.1.6.0.4.

5.2 Apresentação e análise dos resultados

5.2.1 Standard TPC-C

Nesta secção apresentam-se os resultados obtidos usando a carga de testes *standard* do TPC-C. A carga de testes *standard* do TPC-C é composta maioritariamente por transacções de escrita. Existem cinco tipos de transacções que se encontram distribuídas da seguinte maneira: pagamento de encomenda 43%, monitorização de *stock* 4%, entrega de encomendas 4%, consulta do estado 4% e nova encomenda 45%. As transacções de monitorização de *stock* e consulta do estado constituem 8% de transacções *read-only*. Esta distribuição permite avaliar o desempenho do sistema na presença de carga composta maioritariamente por transacções de leitura e escrita.

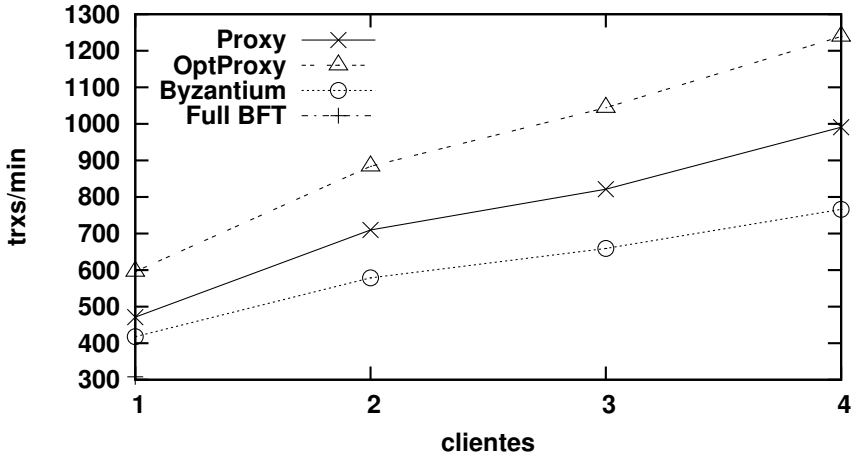

Figura 5.3 Desempenho dos sistemas usando uma carga de trabalho *read-write* (em transacções por minuto).

Na tabela 5.1 apresenta-se a taxa de transacções bem sucedidas durante a execução. No *benchmark* existe à partida uma pequena percentagem de transacções que é abortada sempre. Como era esperado, a taxa de transacções bem sucedidas diminui com o aumento do número de clientes que acedem à base de dados, devido aos problemas de concorrência. Esta diminuição verifica-se em todas as soluções. Analisando os resultados, é possível observar que para cada um dos casos estudados, a diferença na taxa de transacções abortadas é inferior a um por cento. Este resultado permite concluir que, para esta carga de teste, a complexidade adicional do sistema Byzantium não aparenta ter impacto no sucesso das transacções, mesmo utilizando uma carga de testes dominada por transacções de escrita, tipicamente com tempos de execução superiores.

	# Clientes			
Solução	1	2	3	4
Proxy	99.717	86.891	78.994	71.015
OptProxy	99.665	88.084	77.760	69.578
Byzantium	99.761	85.338	77.632	70.226
Full BFT	99.784	-	-	-

Tabela 5.1 Taxa de sucesso de transacções com a carga TPC-C *standard* (em %).

No entanto, é de esperar que com o aumento do número de clientes e com tempos de execução das transacções ainda mais elevados resultem no aumento da taxa de transacções abortadas.

Proxy vs. *OptProxy*

Como se pode ver na tabela 5.2 e na Figura 5.3, a solução *OptProxy* é sempre superior à solução *Proxy*. O tempo de execução de qualquer tipo de transacção é inferior em qualquer cenário de clientes. Como era previsível, a diferença entre as duas soluções baseadas em *proxies* é significativa (cerca de 25%). Este resultado deve-se à diferença entre as implementações. As duas versões distinguem-se em dois aspectos. Primeiro, a versão optimizada não efectua a operação *close* de um ResultSet remotamente, o que permitiu eliminar uma parte considerável das mensagens enviadas. Segundo, sempre que existe um ResultSet envolvido, o sistema procura adiantar informação acerca dos resultados obtidos. Assim, após a execução de uma operação de leitura, um pequeno conjunto de linhas é devolvido ao cliente, de modo a tornar as iterações seguintes independentes da comunicação. Como a grande maioria dos resultados destas operações tem uma dimensão reduzida, o sistema consegue reduzir mais uma parte considerável das mensagens trocadas.

OptProxy vs. Byzantium

Os resultados obtidos revelam que o desempenho do protótipo criado é aproximadamente 30-40% mais lento que a solução *OptProxy*, como se pode verificar na Figura 5.3.

Existem duas razões que podem estar na origem deste *overhead*. Primeiro, devido ao facto de a carga de testes *standard* ser constituída maioritariamente por transacções de escrita. Ao contrário da replicação tradicional, onde apenas se pretende tolerar falhas por omissão, na replicação bizantina, todas as operações de escrita de todas as transacções devem se executadas em

		# Clientes			
Solução	Tipo Trx	1	2	3	4
Proxy	0 (Nova Encomenda)	203	272	349	380
	1 (Pagamento)	49	67	94	109
	2 (Estado)	35	52	59	62
	3 (Entrega)	354	443	532	584
	4 (*Stock*)	13	18	21	29
OptProxy	0 (Nova Encomenda)	157	217	278	300
	1 (Pagamento)	41	58	76	92
	2 (Estado)	19	23	26	30
	3 (Entrega)	255	355	401	487
	4 (*Stock*)	9	11	17	15
Byzantium	0 (Nova Encomenda)	208	300	392	460
	1 (Pagamento)	78	123	170	192
	2 (Estado)	18	23	24	29
	3 (Entrega)	456	495	598	615
	4 (*Stock*)	8	19	16	20
Full BFT	0 (Nova Encomenda)	301	-	-	-
	1 (Pagamento)	82	-	-	-
	2 (Estado)	29	-	-	-
	3 (Entrega)	584	-	-	-
	4 (*Stock*)	12	-	-	-

Tabela 5.2 Tempo médio de execução de cada tipo de transacção usando a carga de trabalho TPC-C *standard* (em ms).

todas as réplicas. Por esta razão, o facto de existirem mais réplicas não permite reduzir a carga em cada uma delas, o que apenas prejudica o resultado final devido à complexidade introduzida pelo algoritmo de replicação, isto é, maior número de mensagens para processar.

Por outro lado, no sistema Byzantium existe a optimização descrita em 3.6.6, que permite evitar a execução das operações de leitura iniciais de escrita em parte das réplicas. No entanto, as transacções de escrita existentes na carga *standard* do *benchmark* TPC-C possuem prefixos *read-only* bastante pequenos, como por exemplo, as transacções de registo de uma nova encomenda que podem chegar às setenta operações, têm apenas duas operações de leitura iniciais.

Assim, tendo em consideração as características da carga utilizada para os testes, tal como as propriedades dos algoritmos de replicação bizantina, o *overhead* verificado encontra-se dentro do esperado e é minimamente aceitável. No entanto, é esperado que este *overhead* possa ser ligeiramente reduzido através de algumas optimizações do código.

Byzantium vs. *Full BFT*

Quando comparando os resultados obtidos durante a execução do Byzantium e a solução na qual todas as operações são executadas pela biblioteca BFT, é possível verificar uma melhoria no desempenho de cerca de 35%. Este resultado diz respeito apenas à situação em que se utiliza um cliente. Não foi possível recolher dados relativos à execução com múltiplos clientes porque ocorrem bloqueios. Esta situação era esperada, uma vez que não foi implementado nenhum mecanismo na solução *Full BFT* que prevenisse o bloqueio de uma operação no acesso à base de dados, tal como o que existe no Byzantium.

Basicamente existem duas diferenças nos algoritmos destas duas soluções que contribuem para a diferença nos resultados. Por um lado, e como a carga de testes é predominantemente composta por transacções de escrita, no sistema Byzantium o tempo total de execução de uma transacção é composto pelo tempo de execução das operações na réplica responsável durante a execução da transacção e pelo tempo necessário para executar as mesmas operações nas réplicas secundárias durante a operação de commit. Isto não acontece na solução *Full BFT* porque todas as operações são executadas em todas as réplicas durante a execução da transacção, e no momento do *commit* essas operações não constituem nenhum atraso adicional. Por outro lado, a solução *Full BFT*, por recorrer à biblioteca de replicação bizantina, necessita de várias rondas de mensagens entre as réplicas do sistema para executar cada uma das operações. Este *overhead* revela-se determinante no mau desempenho desta solução, acabando por contrapor a primeira

diferença indicada.

5.2.2 Carga read-only

Nesta secção apresenta-se uma avaliação do desempenho do sistema para carga de trabalho apenas com transacções de leitura. A carga *read-only* utilizada nos testes consiste em executar apenas as transacções de leitura definidas no *benchmark* TPC-C - consulta de estado e monitorização de *stock*. Aproximadamente 50% de cada tipo de transacção foi executado em cada experiência.

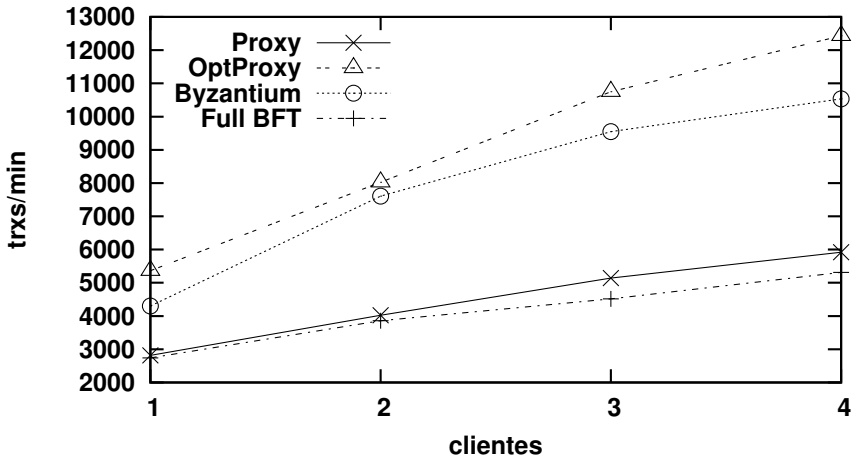

Figura 5.4 Desempenho dos sistemas usando uma carga de trabalho *read-only* (em transacções por minuto).

Proxy vs. *OptProxy*

Ao analisar a tabela na Tabela 5.3, podemos observer que a solução *OptProxy* aumenta a sua vantagem em relação à versão sem optimizações, quando utilizada uma carga de trabalho estritamente só com leituras. A justificação para este acontecimento é semelhante à situação de carga de testes com transacções de leitura e escrita (secção 5.2.1). Esta melhoria dos resultados é justificada pelo facto de as optimizações serem todas elas relacionadas com operações de leitura: iterar e fechar um conjunto de resultados. Assim, não é surpreendente que para esta carga de testes, a solução *OptProxy* apresenta um desempenho superior em cerca de 100%, em todos os casos de estudo.

		# Clientes			
Solução	Tipo Trx	1	2	3	4
Proxy	2 (Estado)	31	43	51	60
	4 (*Stock*)	11	16	19	21
OptProxy	2 (Estado)	14	19	22	25
	4 (*Stock*)	7	10	11	13
Byzantium	2 (Estado)	17	20	24	29
	4 (*Stock*)	10	11	13	15
Full BFT	2 (Estado)	29	42	53	61
	4 (*Stock*)	14	20	26	20

Tabela 5.3 Tempo médio de execução de cada tipo de transacção usando uma carga de trabalho *read-only* (em ms).

OptProxy vs. Byzantium

Após análise dos resultados obtidos, verificou-se que o sistema Byzantium apresentou melhores resultados que a solução *Proxy*. No entanto, os resultados do Byzantium ficam 10-15% abaixo do conseguido pela solução *OptProxy*, o que se justifica por o sistema Byzantium ainda executar algumas operações BFT e pelo fraco impacto do mecanismo de distribuição de carga nas condições dos testes. Este último facto deve-se às máquinas ainda se encontrarem sem problemas de carga com o reduzido número de clientes usados. Para uma avaliação mais eficaz dos mecanismos implementados será necessário usar mais clientes - na próxima secção discutem-se os problemas que levaram a não ter sido possível efectuar esses testes.

Ainda assim, os resultados obtidos permitem verificar que o mecanismo para execução imediata do *commit* no cliente permite melhorar o tempo de execução da operação de *commit*: 2,6 ms no Byzantium e 3,1 ms no *OptProxy*. Estes resultados reflectem que nem sempre é possível confirmar imediatamente a transacção localmente, o que se deve ao facto do *benchmark* utilizado executar uma sequência de operações na base de dados sem nenhum processamento local, o que faz com que em muitos casos a operações de *commit* seja invocada antes do cliente receber os resultados necessário de f+1 réplicas.

Adicionalmente, se for efectuada a comparação do desempenho do sistema Byzantium com aquele da solução *Proxy*, onde todas as operações são propagadas para a réplicas, verifica-se uma melhoria no desempenho de cerca de 75

Byzantium vs. *Full BFT*

Os resultados obtidos mostram que a solução Byzantium tem um desempenho superior à solução *Full BFT*, como era esperado. Essa diferença deve-se principalmente, ao facto de todas as operações serem serializadas pela biblioteca de replicação bizantina, eliminando qualquer possibilidade de execução concorrente das mesmas.

Além disso, a carga introduzida no benchmark é composta exclusivamente por transacções só de leitura. Esse facto constitui uma vantagem para o sistema Byzantium, uma vez que tem a possibilidade de efectuar distribuição da carga. Na realidade o que acontece é que diferentes clientes executam as operações em conjuntos de réplicas diferentes, dando origem a execução paralela real dentro do sistema. Nesta configuração de testes, onde f=1, o conjunto de réplicas que executam uma transacção só com leituras corresponde a metade das réplicas no sistema. Ao distribuir os clientes pelas réplicas, a carga em cada uma das réplicas é reduzida para cerca de metade.

Como foi referido em 5.2.1, a diferença nos algoritmos no momento do *commit* acaba por se dissipar neste caso em que a carga é composta por transacções só com leituras porque a operação de *commit* não exige a execução das operações no momento do *commit* nas réplicas secundárias (porque são executadas durante a execução da transacção). As duas vantagens indicadas anteriormente, aliadas ao facto de não haver esta penalização no momento do *commit* justificam a diferença superior a 90% entre as duas soluções, para número de clientes superiores, como se pode observar na Figura 5.4.

5.2.3 Conclusões

Como era esperado, para uma carga de trabalho composta principalmente por transacções com escritas, o *overhead* existente no sistema Byzantium é considerável quando comparado com o resultado obtido pela solução *OptProxy* (cerca de 30-40%). No entanto, é de referir que esta solução não disponibiliza tolerância a falhas bizantinas.

Por outro lado, o Byzantium revelou um bom desempenho na execução de uma carga de trabalho estritamente *read-only*, reduzindo o *overhead* relativo à solução *OptProxy* para cerca de 10-15%. Como foi justificado neste capítulo, este resultado deve-se à distribuição de carga e ao paralelismo real existente durante a execução das transacções obtido através das optimizações implementadas nesta nova versão do sistema. Note-se que o Byzantium tem à disposição quatro cópias da base de dados enquanto que a solução *OptProxy* apenas acede a uma réplica. Por outro

lado, o Byzantium fornece tolerância a falhas bizantinas.

Após a avaliação inicial apresentada neste capítulo, seria interessante avaliar o protótipo desenvolvido em ambientes com um número de clientes superior ao utilizado, de modo a apurar a escalabilidade do mesmo. No âmbito deste trabalho acabou por não ser possível efectuar estes testes porque no protótipo actual os clientes do *benchmark* têm de executar num processo independente e, na configuração usada, a carga da máquina em que executava os clientes começava a ser considerável com apenas quatro clientes - em grande parte devido à memória usada por cada máquina virtual. Para permitir executar vários clientes no contexto do mesmo processo são necessárias alterações no código do *benchmark* e dos módulos de comunicações, o que não foi possível concluir em tempo útil.

Para complementar os testes realizados ao protótipo, também faria sentido avaliar o peso computacional do sistema, incluindo criptografia.

6. Conclusões e Trabalho Futuro

Esta dissertação apresenta o desenho e a implementação do protótipo da segunda versão do sistema Byzantium. O sistema Byzantium é um sistema de *middleware* de replicação de bases de dados tolerante a falhas bizantinas. Este sistema fornece a semântica de isolamento *Snapshot Isolation*. O sistema é composto por um conjunto de clientes e servidores. Os servidores têm associada uma base de dados que mantém uma réplica completa dos dados. No protótipo actual, usou-se o sistema de gestão de bases de dados PostgreSQL. O código do cliente implementa a interface JDBC permitindo às aplicações aceder ao sistema sem qualquer modificação.

O sistema Byzantium tira partido de um componente de replicação bizantina como base para garantir a convergência eventual entre réplicas. Para efectuar a comunicação entre os componentes do sistema, foi implementado um módulo de comunicação que permite comunicação em grupo e ponto-a-ponto com a semântica FIFO. O sistema usa estes mecanismos de comunicação para executar as transacções efectuadas pelas aplicações num conjunto de réplicas. A aproximação base implementada pelos algoritmos propostos consiste em executar as operações de início e fim da transacção recorrendo à biblioteca de replicação bizantina, enquanto as outras operações são executas especulativamente em apenas uma réplica. Desta forma, o sistema permite que estas operações não sejam afectadas pelo pesado algoritmo de replicação implementado pela biblioteca BFT. A validação da execução de uma transacção é adiada até ao momento do *commit*. Neste momento, as operações executadas com recurso à biblioteca BFT garantem que em todas as réplicas correctas a validação se processa da mesma forma, obtendo o mesmo resultado. Caso os resultados especulativos observados pelo cliente sejam válidos, a transacção termina com sucesso. Caso contrário, aborta.

Em relação à versão anterior do sistema, três modificações principais foram introduzidas nos algoritmos do sistema. Primeiro, o modo como as operações são propagadas. No algoritmo original, o cliente enviava a lista de operações no momento do *commit*. Na nova versão do algoritmo, o cliente envia a operação para todas as réplicas no sistema, através da primitiva de comunicação baseada em *multicast*. Assim, no momento do *commit* não é necessário enviar a lista de operações para confirmar a sua correcção, diminuindo substancialmente a dimensão dessa mensagem. Segundo, a diminuição do número de réplicas que executa uma transacção só com leituras (ou os prefixos de leitura de uma transacção de escrita). Caso não existem réplicas falhadas, é possível executar operações de leitura em apenas f+1 réplicas garantindo a correcção dos resultados (antes executadas em 3f+1 réplicas). Por último, tirando partido

do facto de as operações serem propagadas para as réplicas, é possível executar operações de leitura no momento em que as operações chegam às réplicas com o objectivo de colocar ao dispor do cliente f+1 cópias do resultado no momento do *commit*. Esta alteração permite o retorno imediato das operações de *commit* das transacções *read-only*, no caso normal.

O sistema foi testado com recurso a uma implementação *open-source* do *benchmark standard* TPC-C. Como sistemas de comparação utilizaram-se as seguintes soluções: *Proxy*, um sistema não tolerante a falhas bizantinas onde os clientes propagam as operações para uma réplica que as executa e devolve o resultado; *OptProxy*, semelhante ao anterior mas combinado com um conjunto de optimizações efectuadas no código do protótipo do Byzantium; e *Full BFT*, uma solução que disponibiliza replicação de base de dados tolerante a falhas bizantinas ao executar a totalidade das operações através do componente de replicação BFT. Quando comparado com as versões de *proxy*, a avaliação realizada mostra alguma penalização (30-40%) na utilização do sistema Byzantium com uma carga maioritariamente composta por transacções de escrita. Esta penalização era esperada devido ao facto de as operações terem de executar em todas as réplicas e ser óbvio que os algoritmos implementados impõem alguma sobrecarga na execução das operações. No entanto, esta sobrecarga é necessária para passar de um sistema não tolerante a falhas bizantinas para um sistema tolerante a falhas bizantinas. Os resultados obtidos com a carga de trabalho *read-only* mostram que as alterações propostas neste trabalho permitem uma melhoria no desempenho, tirando partido da distribuição de carga e execução imediata e paralela de transacções de múltiplos clientes. Estas alterações tiveram impacto no *overhead* observado para uma carga de trabalho só com leituras, reduzindo-o para cerca de 10-15% quando comparado com a solução *OptProxy*. Por sua vez, quando comparado com a versão *Proxy* sem optimizações, o desempenho é cerca de 75% melhor.

6.1 Trabalho futuro

No desenvolvimento deste trabalho identificaram-se vários aspectos que podem ser melhorados no futuro. Um aspecto a considerar é a execução de uma avaliação mais completa, de forma a aferir os benefícios dos mecanismo implementados e a escalabilidade do sistema. Faz sentido submeter o sistema a cenários com um número de clientes bastante superior por forma a averiguar o seu comportamento, nomeadamente no que diz respeito à taxe de transacções abortadas. Além disso é importante fazer uma avaliação do peso computacional do sistema, incluindo os

processos de criptografia. Neste momento não foi possível efectuar essa avaliação devido a algumas limitações na implementação que levavam à necessidade de executar cada cliente num processo separado.

Neste domínio da replicação bizantina, existem várias soluções que tiram partido da diversidade de implementações como forma de detecção de erros de implementação. Ao utilizar sistemas de gestão de bases de dados diferentes, será possível diminuir o grau de correlação entre falhas no código, aumentando assim a fiabilidade do sistema. Seria interessante suportar esta funcionalidade no sistema desenvolvido.

Outra das possíveis melhorias seria a eliminação da necessidade de uma operação de início de transacção através da biblioteca de replicação bizantina, pelo menos para as transacções de leitura. Esta solução pode ser implementada com recurso a uma registo de versões da base de dados, mantido no componente Byzantium da réplica. Identificando a versão em que as operações da transacção foram executadas na réplica responsável permite executá-las mais tarde nas réplicas secundárias. Note-se que esta alteração eliminaria a necessidade de usar a biblioteca de replicação bizantina em transacções só com leituras.

Na arquitectura do sistema implementado, o componente de replicação bizantina é visto como uma caixa negra. Nesse sentido, será possível substituí-lo por outra biblioteca de replicação que garanta as mesmas propriedades. O objectivo dessa substituição será identificar implementações desse componente que melhor se adeqúem à solução desenvolvida.

Bibliografia

[1] Backoff protocols for distributed mutual exclusion and ordering. In *ICDCS '01: Proceedings of the The 21st International Conference on Distributed Computing Systems*, page 11, Washington, DC, USA, 2001. IEEE Computer Society.

[2] Michael Abd-El-Malek, Gregory R. Ganger, Garth R. Goodson, Michael K. Reiter, and Jay J. Wylie. Fault-scalable byzantine fault-tolerant services. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 59–74, New York, NY, USA, 2005. ACM.

[3] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying hash functions for message authentication. pages 1–15. Springer-Verlag, 1996.

[4] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ansi sql isolation levels. In *SIGMOD '95: Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, pages 1–10, New York, NY, USA, 1995. ACM.

[5] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *OSDI '99: Proceedings of the third symposium on Operating systems design and implementation*, pages 173–186, Berkeley, CA, USA, 1999. USENIX Association.

[6] James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira. Hq replication: a hybrid quorum protocol for byzantine fault tolerance. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 177–190, Berkeley, CA, USA, 2006. USENIX Association.

[7] Sameh Elnikety, Steven Dropsho, and Fernando Pedone. Tashkent: uniting durability with transaction ordering for high-performance scalable database replication. In *EuroSys '06: Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, pages 117–130, New York, NY, USA, 2006. ACM.

[8] Sameh Elnikety, Steven Dropsho, and Willy Zwaenepoel. Tashkent+: memory-aware load balancing and update filtering in replicated databases. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 399–412, New York, NY, USA, 2007. ACM.

[9] Sameh Elnikety, Willy Zwaenepoel, and Fernando Pedone. Database replication using generalized snapshot isolation. In *SRDS '05: Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems*, pages 73–84, Washington, DC, USA, 2005. IEEE Computer Society.

[10] R. Guerraoui F. Pedone and A. Schiper. The database state machine approach. In *Journal of Distributed and Parallel Databases and Technology*, pages 71–98, 2003.

[11] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

[12] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The dangers of replication and a solution. In *SIGMOD '96: Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, pages 173–182, New York, NY, USA, 1996. ACM.

[13] V. Stankovic I. Gashi, P. Popov and L. Strigini. On designing dependable services with diverse off-the-shelf sql servers. In *Architecting Dependable Systems II*, pages 191–214. Springer Berlin / Heidelberg, 2004.

[14] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative byzantine fault tolerance. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 45–58, New York, NY, USA, 2007. ACM.

[15] Ramakrishna Kotla and Mike Dahlin. High throughput byzantine fault tolerance. In *DSN '04: Proceedings of the 2004 International Conference on Dependable Systems and Networks*, page 575, Washington, DC, USA, 2004. IEEE Computer Society.

[16] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

[17] Hector Garcia Molina, Frank Pittelli, and Susan Davidson. Applications of byzantine agreement in database systems. *ACM Trans. Database Syst.*, 11(1):27–47, 1986.

[18] David Oppenheimer. Why do internet services fail, and what can be done about it? Technical report, Berkeley, CA, USA, 2002.

[19] Christian Plattner and Gustavo Alonso. Ganymed: scalable replication for transactional web applications. In *Middleware '04: Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*, pages 155–174, New York, NY, USA, 2004. Springer-Verlag New York, Inc.

[20] Nuno Preguiça, Rodrigo Rodrigues, Cristovão Honorato, and João Lourenço. Byzantium: Byzantine-fault-tolerant database replication providing snapshot isolation. In *Fourth Workshop on Hot Topics in System Dependability (HotDep '08)*, 2008.

[21] Nicolas Schiper, Rodrigo Schmidt, and O Pedone. Optimistic algorithms for partial database replication. In *In 10th International Conference on Principles of Distributed Systems (OPODIS'2006*, pages 81–93, 2006.

[22] Avi Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill, 2005.

[23] Ben Vandiver, Hari Balakrishnan, Barbara Liskov, and Sam Madden. Tolerating byzantine faults in transaction processing systems using commit barrier scheduling. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 59–72, New York, NY, USA, 2007. ACM.

[24] Jian Yin, Jean-Philippe Martin, Arun Venkataramani, Lorenzo Alvisi, and Mike Dahlin. Separating agreement from execution for byzantine fault tolerant services. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 253–267, New York, NY, USA, 2003. ACM.